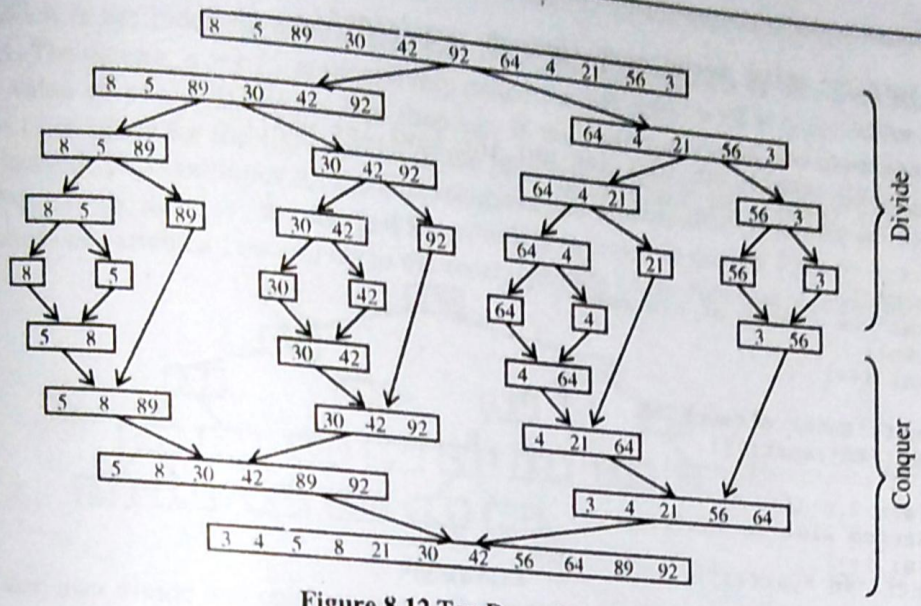


Figure 8.12 Top Down Merge Sort

- arr[0:10] is divided into arr[0:5] and arr[6:10]
  (8, 5, 89, 30, 42, 92, 64, 4, 21, 56, 3) -> (8, 5, 89, 30, 42, 92) and (64, 4, 21, 56, 3)
  - arr[0:5] is divided into arr[0:2] and arr[3:5]
    (8, 5, 89, 30, 42, 92) -> (8, 5, 89) and (30, 42, 92)
    - arr[0:2] is divided into arr[0:1] and arr[2]
      (8, 5, 89) -> (8, 5) and (89)
      - arr[0:1] is divided into arr[0] and arr[1]
        (8, 5) -> (8) and (5)
      - arr[0] and arr[1] are merged to produce sorted arr[0:1]
        (8) and (5) -> (5, 8)
    - arr[0:1] and arr[2] are merged to produce sorted arr[0:2]
      (5, 8) and (89) -> (5, 8, 89)
    - arr[3:5] is divided into arr[3:4] and arr[5]
      (30, 42, 92) -> (30, 42) and (92)
      - arr[3:4] is divided into arr[3] and arr[4]
        (30, 42) -> (30) and (42)
      - arr[3] and arr[4] are merged to produce sorted arr[3:4]
        (30) and (42) -> (30, 42)
    - arr[3:4] and arr[5] are merged to give sorted arr[3:5]
      (30, 42) and (92) -> (30, 42, 92)
  - arr[0:2] and arr[3:5] are merged to produce sorted arr[0:5]
    (5, 8, 89) and (30, 42, 92) -> (5, 8, 30, 42, 89, 92)
  - arr[6:10] is divided into arr[6:8] and arr[9:10]
    (64, 4, 21, 56, 3) -> (64, 4, 21) and (56, 3)
    - arr[6:8] is divided into arr[6:7] and arr[8]
      (64, 4, 21) -> (64, 4) and (21)
      - arr[6:7] is divided into arr[6] and arr[7]
        (64, 4) -> (64) and (4)
      - arr[6] and arr[7] are merged to produce sorted arr[6:7]
        (64) and (4) -> (4, 64)
    - arr[6:7] and arr[8] are merged to produce sorted arr[6:8]
      (4, 64) and (21) -> (4, 21, 64)
    - arr[9:10] is divided into arr[9] and arr[10]
      (56, 3) -> (56) and (3)
    - arr[9] and arr[10] are merged to produce sorted arr[9:10]
      (56) and (3) -> (3, 56)
  - arr[6:8] and arr[9:10] are merged to produce sorted arr[6:10]
    (4, 21, 64) and (3, 56) -> (3, 4, 21, 56, 64)
- arr[0:5] and arr[6:10] are merged to produce sorted arr[0:10]
  (5, 8, 30, 42, 89, 92) and (3, 4, 21, 56, 64) -> (3, 4, 5,8, 21, 30, 42, 56, 64, 89, 92)

```
/*P8.6 Program of sorting using merge sort through recursion*/
#include<stdio.h>
#define MAX 100
void merge_sort(int arr[],int low,int up);
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2);
void copy(int arr[],int temp[],int low,int up);
main()
{
        int i,n,arr[MAX];
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        merge_sort(arr,0,n-1);
        printf("\nSorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
}/*End of main()*/
void merge_sort(int arr[],int low,int up)
{
        int mid;
        int temp[MAX];
        if(low<up)   /*if more than one element*/
        {
                mid = (low+up)/2;
                merge_sort(arr,low,mid);  /*Sort arr[low:mid]*/
                merge_sort(arr,mid+1,up);  /*Sort arr[mid+1:up]*/
                /*Merge arr[low:mid] and arr[mid+1:up] to temp[low:up]*/
                merge(arr,temp,low,mid,mid+1,up);
                copy(arr,temp,low,up); /* Copy temp[low:up] to arr[low:up] */

        }
}/*End of merge_sort*/
/*Merges arr[low1:up1] and arr[low2:up2] to temp[low1:up2]*/
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2)
{
        int i = low1;
        int j = low2 ;
        int k = low1 ;
        while((i<=up1) && (j<=up2))
        {
                if(arr[i] <= arr[j])
                        temp[k++] = arr[i++] ;
                else
                        temp[k++] = arr[j++] ;
        }
        while(i<=up1)
                temp[k++] = arr[i++];
        while(j<=up2)
                temp[k++] = arr[j++];
}/*End of merge()*/
void copy(int arr[],int temp[],int low,int up)
{
        int i;
        for(i=low; i<=up; i++)
                arr[i] = temp[i];
}
```

The process is recursive so the record of the lower and upper bound of the sublists is implicitly maintained. When `merge_sort()` is called for the first time, the `low` and `up` are set to 0 and `n-1`. The value of `mid` is

calculated which is the index of middle element. Now the `merge_sort()` is called for left sublist which is `arr[0:mid]`. The `merge_sort()` is recursively called for left sublists till it is called for a one element sublist. In this call, value of `low` will not be less than up, it will be equal to up so recursion will terminate. Now `merge_sort()` is called for the right sublist of the previous recursive call. After the return of this call, the two sublists are merged. This continues till the first recursive call returns after which we get the sorted result. Before returning from `merge_sort()`, the merged list which is in `temp` is copied back to `arr`. The following figure shows the values of variables `low` and up in the recursive calls of the function `merge_sort()`.

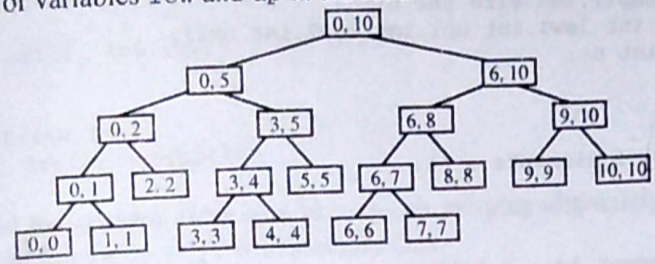


Figure 8.13

Quick sort also uses divide and conquer approach but there partition step is difficult while combining is trivial, here in merge sort partition is simple but combining is difficult.

8.12.2 Analysis of Merge Sort

We know that n elements can be repeatedly divided into half approximately $\log_2 n$ times, so after halving the list $\log_2 n$ times we get n sublists of size 1. In each pass there will be merging of n elements which is O(n) so performance of merge sort is $O(n\log_2 n)$.

Merge sort is a stable sort. Since the operation of merging is not in place, merge sort is also not an in place sort and requires O(n) extra space.

8.12.3 Bottom Up Merge Sort(Iterative)

The stack space needed for recursion can be avoided by implementing a non recursive version of merge sort. The iterative version works in bottom up manner and uses combine and conquer strategy.

In this approach, the whole list is initially considered as n sorted sublists of size 1. The adjacent sublists of size 1 are merged pairwise and we get n/2 sublists of size 2 (and possibly one more sublist of size 1 if n is odd). These sublists of size 2 are then merged and we get n/4 sublists of size 4 (and possibly one sublist of size 1, 2 or 3). This process continues until only one sublist of size n is left.

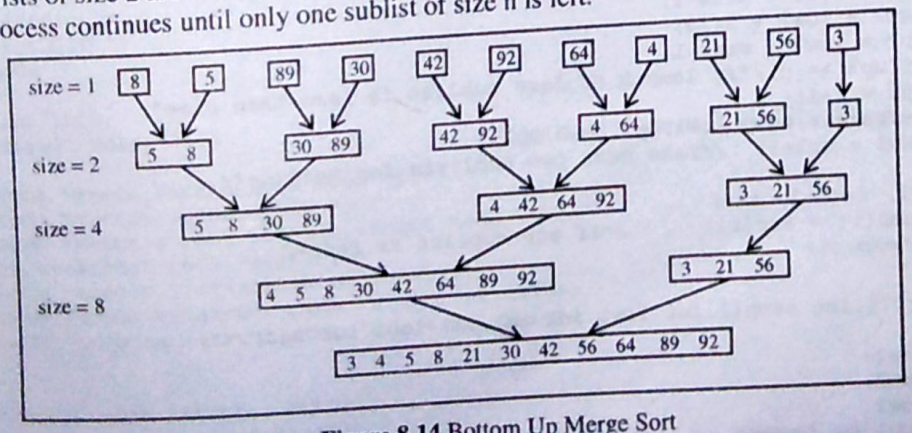


Figure 8.14 Bottom Up Merge Sort

If there is a sublist left in the last which cannot be merged with any sublist, it is just copied to the result. In the example given in figure 8.14, this case occurs in pass 1 where sublist [3] is left alone and in pass 3 where sublist [3, 21, 56] is left alone.

```
/*P8.7 Program of sorting using merge sort without recursion*/
#include<stdio.h>
#define MAX 100
void merge_sort(int arr[],int n);
void merge_pass(int arr[],int temp[],int size,int n);
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2);
void copy(int arr[],int temp[],int n);
main()
{
        int arr[MAX],i,n;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        merge_sort(arr,n);
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
}/*End of main()*/
void merge_sort(int arr[],int n)
{
        int temp[MAX];
        int size = 1;
        while(size<n)
        {
                merge_pass(arr,temp,size,n);
                size = size*2;
        }
}
void merge_pass(int arr[],int temp[],int size,int n)
{
        int i,low1,up1,low2,up2;
        low1 = 0;
        while(low1+size < n)
        {
                up1 = low1 + size-1;
                low2 = low1 + size;
                up2 = low2 + size-1;
                if(up2 >= n)/*if length of last sublist is less than size*/
                up2 = n-1;
                merge(arr,temp,low1,up1,low2,up2);
                low1 = up2+1;  /*Take next two sublists for merging*/
        }
        for(i=low1; i<=n-1; i++)
                temp[i] = arr[i];       /*If any sublist is left*/
        copy(arr,temp,n);
}
void merge(int arr[],int temp[],int low1,int up1,int low2,int up2)
{
        int i = low1;
        int j = low2;
        int k = low1;
        while(i<=up1 && j<=up2)
        {
```



```
                if(
                els
        }
        while(i<=u
                ter
        while(j<=u
                ter
}
void copy(int arr
{
        int i;
        for(i=0;i<
                ar
}
```

We can avoid th
arr. The function m

```
merge_sort(int ar
{
        int temp[
        while(siz
        {
                m
                s
                m
                s
        }
}
```

Now we can de
arr. The total nu
$O(n\log_2 n)$.

8.12.4 Merge

When merge sort
linked lists can be
is no data movem

```
/*P8.8 Program
#include<stdio.
#include<stdli
struct node
{
        int inf
        struct
};
struct node *m
struct node *d
struct node *m
void display(s
struct node *c
struct node *a
struct node *a
main()
{
        struct
        start
        start
        displ
```

```
            if(arr[i] <= arr[j])
                    temp[k++] = arr[i++];
            else
                    temp[k++] = arr[j++];
        }
        while(i<=up1)
                temp[k++] = arr[i++];
        while(j<=up2)
                temp[k++] = arr[j++];
}
void copy(int arr[], int temp[], int n)
{
        int i;
        for(i=0;i<n;i++)
                arr[i] = temp[i];
}
```

We can avoid the copying from `arr` to `temp` by merging alternately from `arr` to `temp` and from `temp` to `arr`. The function `merge_sort()` would be like this-

```
merge_sort(int arr[], int n)
{
        int temp[MAX], size = 1;;
        while(size<n)
        {
                merge_pass(arr,temp,size,n);
                size = size*2;
                merge_pass(temp,arr,size,n);
                size = size*2;
        }
}
```

Now we can delete the last statement from the function `merge_pass()` which is used for copying `temp` to `arr`. The total number of passes required is $\log_2 n$ and in each pass n elements are merged, so complexity is $O(n\log_2 n)$.

8.12.4 Merge Sort for linked List

When merge sort is used for linked lists, there is no need of temporary storage because merge operation in linked lists can be performed without temporary storage. Another advantage with linked list version is that there is no data movement, only the pointers are rearranged.

```
/*P8.8 Program of merge sort for linked lists*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *merge_sort(struct node *p);
struct node *divide(struct node *p);
struct node *merge(struct node *p,struct node *q);
void display(struct node *start);
struct node *create_list(struct node *start);
struct node *addatbeg(struct node *start,int data);
struct node *addatend(struct node *start,int data);
main()
{
        struct node *start = NULL;
        start = create_list(start);
        start = merge_sort(start);
        display(start);
```

```
}
struct node *merge_sort(struct node *start)
{
        struct node *start_first, *start_second, *start_merged;
        if(start!=NULL && start->link!=NULL) /*if more than one element*/
        {
                start_first = start;
                start_second = divide(start);
                start_first = merge_sort(start_first);
                start_second = merge_sort(start_second);
                start_merged = merge(start_first,start_second);
                return start_merged;
        }
        else
                return start;
}
struct node *divide(struct node *p)
{
        struct node *q,*start_second;
        q = p->link->link;
        while(q!=NULL)
        {
                p = p->link;
                q = q->link;
                if(q!=NULL)
                        q = q->link;
        }
        start_second = p->link;
        p->link = NULL;
        return start_second;
}
struct node *merge(struct node *p1,struct node *p2)
{
        struct node *start_merged,*q;
        if(p1->info <= p2->info)
        {
                start_merged = p1;
                p1 = p1->link;
        }
        else
        {
                start_merged = p2;
                p2 = p2->link;
        }
        q = start_merged;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info <= p2->info)
                {
                        q->link = p1;
                        q = q->link;
                        p1 = p1->link;
                }
                else
                {
                        q->link = p2;
                        q = q->link;
                        p2 = p2->link;
                }
        }
        if(p1!=NULL)
                q->link = p1;
```



```
        else
        return
}
void display(s
{       struct
        if(sta
        {
        }
        p = sta
        printf(
        while(p
        {
        }
        printf(
}/*End of disp
struct node *cr
{
        int i,n
        printf(
        scanf("
        start=N
        printf(
        scanf("
        start=a
        for(i=2
        {
        }
        return
}/*End of creat
struct node *ad
{
        struct
        tmp = ma
        tmp->inf
        tmp->lin
        start =
        return s
}/*End of addat
struct node *ad
{
        struct
        tmp = ma
        tmp->inf
        p = star
        while(p-
        p->link
        tmp->lin
        return s
}/*End of addat
```

```
        else
                q->link = p2;
        return start_merged;
}
void display(struct node *start)
{
        struct node *p;
        if(start==NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = start;
        printf("List is :\n");
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p = p->link;
        }
        printf("\n");
}/*End of display()*/
struct node *create_list(struct node *start)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start=NULL;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addatbeg(start,data);
        for(i=2;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start=addatend(start,data);
        }
        return start;
}/*End of create_list()*/
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = start;
        start = tmp;
        return start;
}/*End of addatbeg()*/
struct node *addatend(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = malloc(sizeof(struct node));
        tmp->info = data;
        p = start;
        while(p->link!=NULL)
                p = p->link;
        p->link = tmp;
        tmp->link = NULL;
        return start;
}/*End of addatend()*/
```

The function `divide()` takes a pointer to the original list and returns a pointer to the start of the second sublist. We have taken two pointers p and q, where pointer p points to the first node and pointer q points to the third node. In a loop we move pointer p once and pointer q twice so that when the pointer q is NULL, pointer p points to the middle node. The node next to the middle node will be the start of second sublist. The middle node will be the last node of first sublist so its link is assigned NULL.

The function `merge()` takes pointers to the two sorted lists, merges them into a single sorted list and returns a pointer to the merged list. There is no requirement of any temporary storage for merging of linked lists.

8.12.5 Natural Merge Sort

Merge sort does not consider any sorted sequences inside a given list and sorts in usual manner. This type of merge sort is called straight merge sort, and in this sort all the sublists in a pass are of the same size except possibly the last one. Another type of merge sort is natural merge sort which takes advantage of the presence of sorted sequences(runs) in the list. For example consider this list-

12 45 67 3 66 21 89 90 98 65 43 56 68 96 23 87

The runs present in this list are (12, 45, 67) (3, 66) (21, 89, 90, 98) (65) (43, 56, 68, 96) (23, 87). In the first pass the runs are determined and these runs are merged instead of merging sublists of size 1, remaining passes are same as in the merge sort.

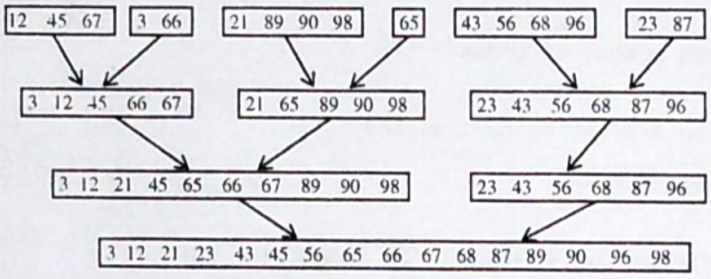


Figure 8.14 Natural Merge Sort

The disadvantage with this approach is that the size of sublists is different and we have to keep track of the start and end of each sublist. So this technique is beneficial only if there are many runs present in the data. Merge sort is not suitable for main memory sorts because it requires extra space of O(n) but this algorithm forms the basis of external sorting techniques. Merge sort is stable because the merge process is stable.

8.13 Quick Sort (Partition Exchange Sort)

Quick Sort was given by C.A.R. Hoare in 1962 and is based on the divide and conquer technique. In this technique, a problem is divided into small problems which are again divided into smaller problems and so on, hence solving the bigger problem reduces to solving these smaller ones. As the name suggests, quick sort is a very fast sorting technique and its average case performance is O(n log n). It is an in-place sort so no additional space is required for sorting. It is also known as partition exchange sort and is very efficient because the exchanges occur between elements that are far apart, and so less exchanges are needed to place an element in its final position.

Now let us see how this sorting is performed. We choose an element from the list and place it at its proper position in the list, i.e. at the position where it would be in the final sorted list. We call this element as pivot and it will be at its proper place if-

(i) all the elements to the left of pivot are less than or equal to the pivot and

(ii) all the elements to the right of pivot are greater than or equal to the pivot.

Any element of the list can be taken as pivot but for convenience the first element is taken as the pivot, we will talk more about the choice of pivot later. Suppose our list is [4, 6, 1, 8, 3, 9, 2, 7], if we take 4 as the pivot

then after placing 4 at its proper place the list becomes [1, 3, 2, **4**, 6, 8, 9, 7]. Now we can partition this list into two sublists based on this pivot and these sublists are [1, 3, 2] and [6, 8, 9, 7]. We can apply the same procedure to these two sublists separately i.e. we will select one pivot for each sublist and both the sublists are divided into 2 sublists each so now we get 4 sublists. This process is repeated for all the sublists that contain two or more elements and in the end we get our sorted list.

Now let us outline the process for sorting the elements through quick sort. Suppose we have an array `arr` with `low` and `up` as the lower and upper bounds.

1. Take the first element of list as pivot.
2. The list is partitioned in such a way that pivot comes at its proper place. After this partition, all elements to the left of pivot are less than or equal to the pivot and all elements to the right of pivot are greater than equal to the pivot. So one element of the list i.e. pivot is at its proper place. Let the index of pivot be `pivotloc`.
3. Create two sublists left and right side of pivot, left sublist is `arr[low].....arr[pivotloc-1]` and the right sublist is `arr[pivotloc+1]....arr[up]`.
4. The left sublist is sorted using quick sort recursively.
5. The right sublist is sorted using quick sort recursively.
6. The terminating condition for recursion is - when the sublist formed contains only one element or no element.

The sublists are kept in the same array, and there is no need of combining the sorted sublists at the end. Let us take a list of elements and sort them through quick sort.

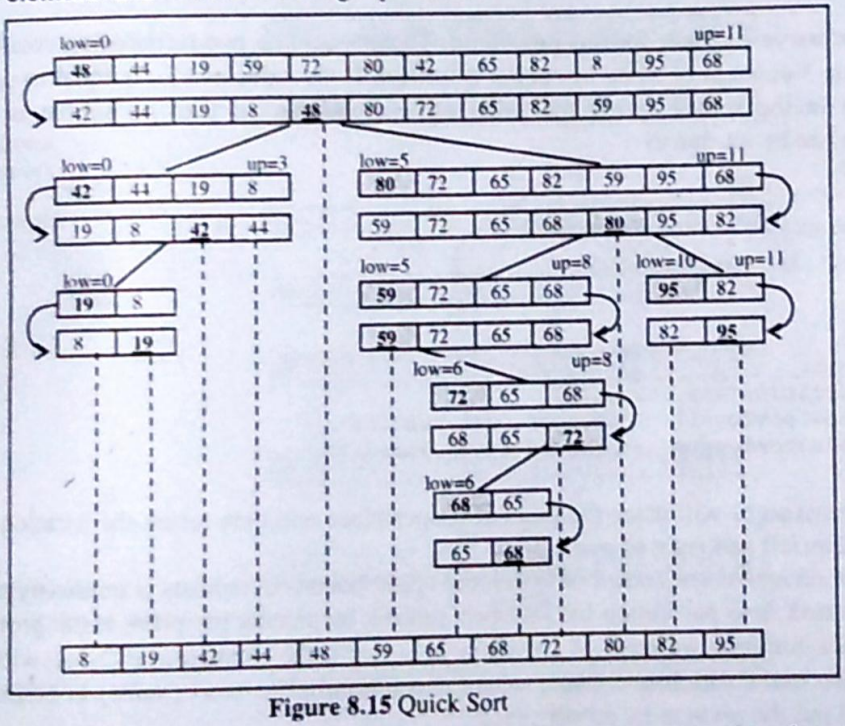


Figure 8.15 Quick Sort

Here we are focusing only on the recursion procedure; the logic of placing the pivot at proper place is discussed later. The values of `low` and up indicate the lower and upper bounds of the sublists.

Initially the list is [48, 44, 19, 59, 72, 80, 42, 65, 82, 8, 95, 68]. We will take the first element(48) as the pivot and place it in proper place and so the list becomes [42, 44, 19, 8, **48**, 80, 72, 65, 82, 59, 95, 68]. Now all the elements to the left of 48 are less than it and all elements to the right of 48 are greater than it. We will take two sublists left and right of 48 which are [42, 44, 19, 8] and [80, 72, 65, 82, 59, 95, 68], and sort them separately using the same procedure. Note that the order of the elements in the left sublist or in the right sublist is not the same as it appears in the original list. It depends on the partition process which is used to place pivot

at proper place. For now you just need to understand that all elements to left of pivot are less than or equal to pivot and all elements right of pivot are greater than or equal to pivot.

The left sublist is [42, 44, 19, 8] and its pivot is taken as 42, after placing the pivot the list becomes [19, 8, **42**, 44]. Now 42 is at its proper place, we again divide this list into two sublists which are [19, 8] and [44]. The list [44] has only one element so we will stop. The list [19, 8] is taken and here the pivot will be 19 and after placing the pivot at proper place the list becomes [8, **19**]. The left sublist is [8] and it contains only one element so we will not process it. There are no elements to the right of 19, so right sublist is not formed or we can say that right sublist contains zero elements so we will stop.

The right sublist is [80, 72, 65, 82, 59, 95, 68] and 80 is taken as the pivot. After placing the pivot the list becomes [59, 72, 65, 68, **80**, 95, 82]. Now the two sublists formed are [59, 72, 65, 68] and [95, 82]. In the first sublist, 59 is taken as the pivot and after placing it at the proper place the sublist becomes [**59**, 72, 65, 68]. There are no elements to the left of 59 so only one sublist is formed which is [72, 65, 68]. Here 72 is taken as the pivot and after placing it the list becomes [68, 65, **72**]. There are no elements to the right of 72 so only one sublist is formed which is [68, 65]. In this sublist 68 is taken as the pivot and after placing it at proper place the sublist becomes [65, **68**]. There are no elements to the right of 68 and the left sublist has only one element so we will stop.

After this we will take the sublist [95, 82]. Here 95 is taken as the pivot and after placing it at proper place the sublist is [82, **95**]. There are no elements to the right of 95 and the left sublist has only one element so we will stop.

We can write a recursive function for this procedure. There would be two terminating conditions, when the sublist formed has only 1 element or when no sublist is formed. If the value of `low` is equal to `up` then there will be only one element in the sublist and if the value of `low` exceeds `up` then no sublist is formed. So the terminating condition can be written as-

```
if(low>=up)
        return;
```

The recursive function `quick()` can be written as-

```
void quick(int arr[],int low,int up)
{
        int pivloc;
        if(low>=up)
                return;
        pivloc = partition(arr,low,up);
        quick(arr,low,pivloc-1); /*process left sublist*/
        quick(arr,pivloc+1,up);  /*process right sublist*/
}/*End of quick()*/
```

The function `partition()` will place the pivot at proper place and then return the location of pivot so that we can form two sublists left and right of pivot.

Since the process is recursive, the record of lower and upper bounds of sublists is implicitly maintained.

Now the second main task is to partition a list into two sublists by placing the pivot at the proper place. Let us see how we can do this. Suppose we have an array `arr[low:up]`, the element `arr[low]` will be taken as the pivot. We will take two index variables `i` and `j` where `i` is initialized to `low+1` and `j` is initialized to up. The following process will put the pivot at its proper place.

(a) Compare the pivot with `arr[i]`, and increment `i` if `arr[i]` is less than the pivot element. So the index variable `i` moves from left to right and stops when we get an element greater than or equal to the pivot.

(b) Now compare the pivot with `arr[j]`, and decrement `j` if `arr[j]` is greater than the pivot element. So the index variable `j` moves from right to left, and stops when we get an element less than or equal to the pivot.

(c) If `i` is less than `j`

Exchange the value of `arr[i]` and `arr[j]`, increment `i` and decrement `j`.

else

No exchange, increment `i`.

(d) Repeat the steps (a), (b), (c) ti
(e) When value of `i` becomes m
now pivot is to be placed at pos
value of `arr[low]` and `arr[j]`.

Now let us take a list and see hov

low=0, up=11
i=1 and j=11. Pivot = arr[low] = 48

44 < 48, Increment i

19 < 48, Increment i

59 > 48, Stop i at 3

68 > 48, Decrement j

95 > 48, Decrement j

8 < 48, Stop j at 9.
Now both i and j stopped
Since i < j, Exchange arr[3] and arr[9]

Increment i and decrement j

72 > 48, stop i at 4

82 > 48, decrement j

65 > 48, decrement j

42 < 48, stop j at 6
Both i and j stopped
Since i < j, Exchange arr[4] and arr[6

Increment i and decrement j

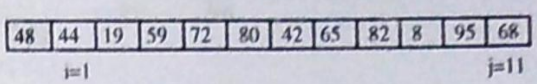
80 > 48, stop i at 5

80 > 48, decrement j

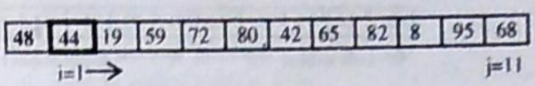
(d) Repeat the steps (a), (b), (c) till the value of `i` is less than or equal to `j`. We will stop when `i` exceeds `j`.
(e) When value of `i` becomes more than `j`, we have found proper place for pivot which is given by `j`. hence now pivot is to be placed at position `j`. Pivot was at location `low`, so we can place it at `j` by exchanging the value of `arr[low]` and `arr[j]`. Now pivot is at position `j` which is its final position.

Now let us take a list and see how pivot will be placed at proper place through this partition process.

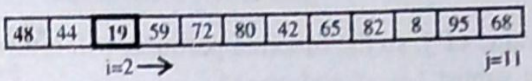
low=0, up=11
i=1 and j=11, Pivot = arr[low] = 48



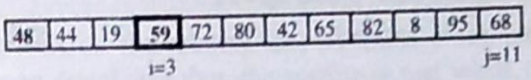
44 < 48, Increment i



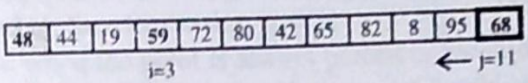
19 < 48, Increment i



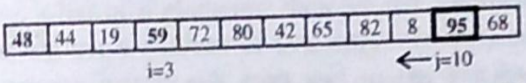
59 > 48, Stop i at 3



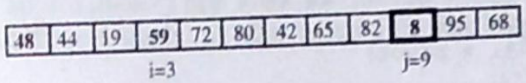
68 > 48, Decrement j



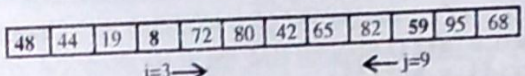
95 > 48, Decrement j



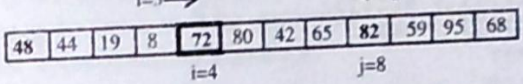
8 < 48, Stop j at 9.
Now both i and j stopped
Since i < j, Exchange arr[3] and arr[9]



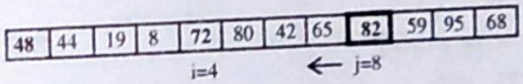
Increment i and decrement j



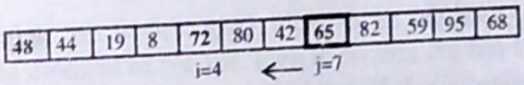
72 > 48, stop i at 4



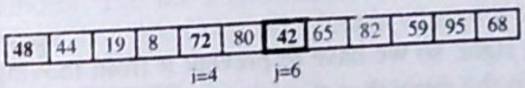
82 > 48, decrement j



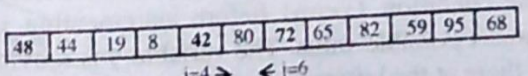
65 > 48, decrement j



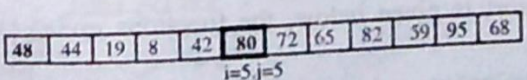
42 < 48, stop j at 6
Both i and j stopped
Since i < j, Exchange arr[4] and arr[6]



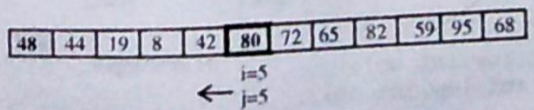
Increment i and decrement j



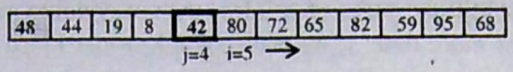
80 > 48, stop i at 5



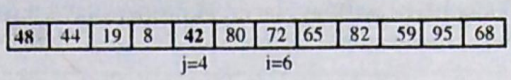
80 > 48, decrement j



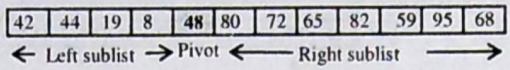
42 < 48, stop j at 4
Both i and j stopped.
i is not less than j, so no exchange
i is incremented



Now i > j, so stop movement of i and j.
j is the location for pivot.
Exchange arr[0] and arr[4]



Pivot placed at proper place



The location for pivot is the value of `j`, so the function partition will return value of `j`. Now for left sublist `low=0` and `up=3` and for right sublist `low=5` and `up=11`.

```
int partition(int arr[],int low,int up)
{
        int temp,i,j,pivot;
        i = low+1;
        j = up;
        pivot = arr[low];
        while(i <= j)
        {
                while((arr[i] < pivot) && (i < up))
                        i++;
                while(arr[j] > pivot)
                        j--;
                if(i < j)
                {
                        temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                        i++;
                        j--;
                }
                else
                        i++;
        }
        arr[low] = arr[j];
        arr[j] = pivot;
        return j;
}/*End of partition()*/
```

The variable `i` is moving right, so we have to prevent it from moving past the array bound. For example if pivot is the largest element in the array then there is no element in the array that can stop `i`, and it just moves on. So we have to check the condition `(i<up)` before incrementing `i`. The other way out may be to put a sentinel at the end having largest possible value. The variable `j` is moving left but it will never cross the bound of the array because pivot is there at the leftmost position.

The program for quick sort is given below, the functions `quick()` and `partition()` are the same as described earlier.

```
/*P8.9 Program of sorting using quick sort*/
#include<stdio.h>
#define MAX 100
void quick(int arr[],int low,int up);
int partition(int arr[],int low,int up);
main()
{
        int array[MAX],n,i;
```

```
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
            printf("Enter element %d : ",i+1);
            scanf("%d",&array[i]);

    }
    quick(array,0,n-1);
    printf("Sorted list is :\n");
    for(i=0; i<n; i++)
            printf("%d ",array[i]);
    printf("\n");
}/*End of main()*/
```

The partition process is not stable so quick sort is not a stable sort.

8.13.1 Analysis of Quick Sort

The time requirement of quick sort depends on the relative size of the two sublists formed. If the partition is balanced and the two sublists are of almost equal size then the sorting is fast while if the partition is unbalanced and one sublist is much larger than the other, then the sorting is slow.

The best case for quick sort would be when the pivot is always placed in the middle of the list, i.e. when we get two sublists of equal size. If we have a list of n elements, then we get 2 sublists of size approximately n/2 which are again divided equally so that we get 4 sublists of size approximately n/4 each and these are again divided and we get 8 sublists of size approximately n/8 each and so on. We know that n elements can be repeatedly divided into half approximately $\log_2 n$ times, so after halving the list $\log_2 n$ times we get n sublists of size 1.

Now let's find out the number of comparisons that will be performed. Initially we have 1 list of size n for which there will be approximately n comparisons (exactly n-1, but we want Oh notation only so we can take n), then we have 2 lists of size approximately n/2 each so there will be approximately 2*(n/2) comparisons, then we have 4 lists of size approximately n/4 each so there will be approximately 4*(n/4) comparisons and so on. So the total number of comparisons would be approximately equal to-

$n + 2*(n/2) + 4*(n/4) + 8*(n/8) + \ldots\ldots\ldots\ldots n*(n/n)$

$n + n + n + \ldots\ldots\ldots\ldots + n$ ($\log_2 n$ terms)

$= n\log_2 n$

So in best case the performance of quick sort is $O(n\log_2 n)$. Now let us see how quick sort performs in worst case. If the pivot is the smallest or largest element of the list, then it divides the list into two sublists from which one is empty and other contains n-1 elements. The worst case occurs if this situation arises in every recursive call. If the first element is taken as the pivot then this worst case occurs when the list is fully sorted(or reverse sorted).

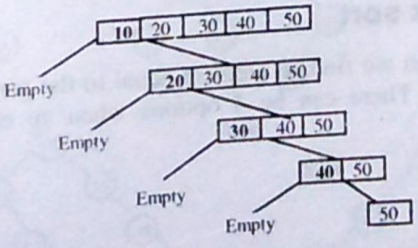


Figure 8.16

If we have a list of n elements, then first we get 2 sublists of sizes 0 and n-1. Now sublist of size n-1 is divided into two sublists of sizes 0 and n-2. The total number of lists that are sorted is n-1 and these are of sizes n, n-1, n-2, ...2. The total number of comparisons will be-

$n-1 + n-2 + n-3 + ... + 1$
$= n(n-1)/2$
$= O(n^2)$

So in worst case the performance of quick sort is $O(n^2)$.

The average case performance is closer to the best case than to the worst case and is found out to be $O(n\log_2 n)$. It is not a stable sort. Space complexity for this sort is $O(\log n)$.

8.13.2 Choice of pivot in Quick Sort

The quick sort performs best when the pivot is chosen such that it divides the list into two equal sublists. It performs badly if the pivot value does an unbalanced partition i.e. one sublist is very small and the other one is big. So the efficiency of quick sort can be improved by choosing a pivot value which makes balanced partitions. The worst case can be avoided by careful selection of the pivot.

The original algorithm given by Hoare selected the first element as the pivot but the first element is not a good choice because if the list is sorted or almost sorted then partitions will be unbalanced and performance will not be good. The last element is also not a good choice for similar reasons. As the chances of a list being sorted or almost sorted are high, it is better to avoid the first and last elements.

Another option is to choose the pivot randomly. For this we chose a random number `k` between `low` and `up`, and take `arr[k]` as the pivot. This `arr[k]` is interchanged with the first element before the while loop in the function `partition()` and rest of the code remains same. It seems to be a safe option but random number generators are time consuming.

The ideal choice would be to take the median value of all the elements but this option is costly so we can use median of three method. In this method we take the median of the first, middle and last elements i.e. median of `{ arr[low], arr[(low+up)/2], arr[up] }` and then interchange the median element with the first element.

```
mid = (low+up)/2;
if(arr[low] > arr[mid])
        exchange(arr[low],arr[mid]);
if(arr[low] > arr[up])
        exchange(arr[low],arr[up]);
if(arr[mid] > arr[right])
        exchange(arr[mid],arr[right]);
exchange(arr[low],arr[mid]);
```

In the process of finding the median we have placed the largest of three at the end, so pivot cannot be greater than the last element and hence now there is no need of condition `i<up` before incrementing `i`.

8.13.3 Duplicate elements in quick sort

In our program we stop variables `i` and `j` when we find an element equal to the pivot. Let us see what other options are and why they are less efficient. There can be 4 options when an element equal to pivot is encountered

(i) stop `i` and move `j`
(ii) stop `j` and move `i`
(iii) stop both `i` and `j`
(iv) move both `i` and `j`

If we stop one pointer and move another then all elements equal to the pivot would go in one sublist and the partitioning will be unbalanced. For example if we stop `i` and move `j`, then all the equal elements go to the right sublist, and if we stop `j` and move `i` then all the equal elements go to the left sublist. The first two options are

not good because they tend to maximize the difference in the sizes of the sublists which reduces the efficiency of quick sort.

To understand which of the last two options is better, let us consider the case when all the elements in the list are equal. If both `i` and `j` stop then there will be many unnecessary exchanges between equal elements but the good thing is that `i` and `j` will meet somewhere in middle of the list thus creating two almost equal sublists. If both `i` and `j` move then no unnecessary swaps would be there but the sublists would be unbalanced. If all the elements are equal then `j` will always stop at the leftmost position and so pivot will always be placed at the leftmost position and hence one sublist will always be empty. This is the same situation that we studied in the worst case and the running time is $O(n^2)$.

So it is best to stop `i` and `j` when any element equal to the pivot is encountered. It might seem that considering the case of all equal elements is not a good idea as it would rarely occur. A list of all equal elements can be rare but we may get a sublist consisting of all equal elements, for example suppose we have a list of 500,000 elements out of which 10,000 are equal. Since quick sort is recursive so at some point there will be a recursive call for these 10,000 identical elements and if sorting these elements takes quadratic time then the overall efficiency will be affected.

8.14 Binary tree sort

Binary tree sort uses a binary search tree for sorting the data. This algorithm proceeds in two phases, namely construction phase and traversal phase.

1. Construction Phase - A binary search tree is created from the given data.
2. Traversal Phase - The binary search tree created is traversed in inorder to obtain the sorted data.

The details of creation and traversal of a binary search tree are discussed in chapter on trees. Let us take an array `arr` containing unsorted elements and sort them using binary tree sort.

19	35	10	12	46	6	40	3	90	8
0	1	2	3	4	5	6	7	8	9

First we will insert all the elements of this array in a binary search tree one by one.

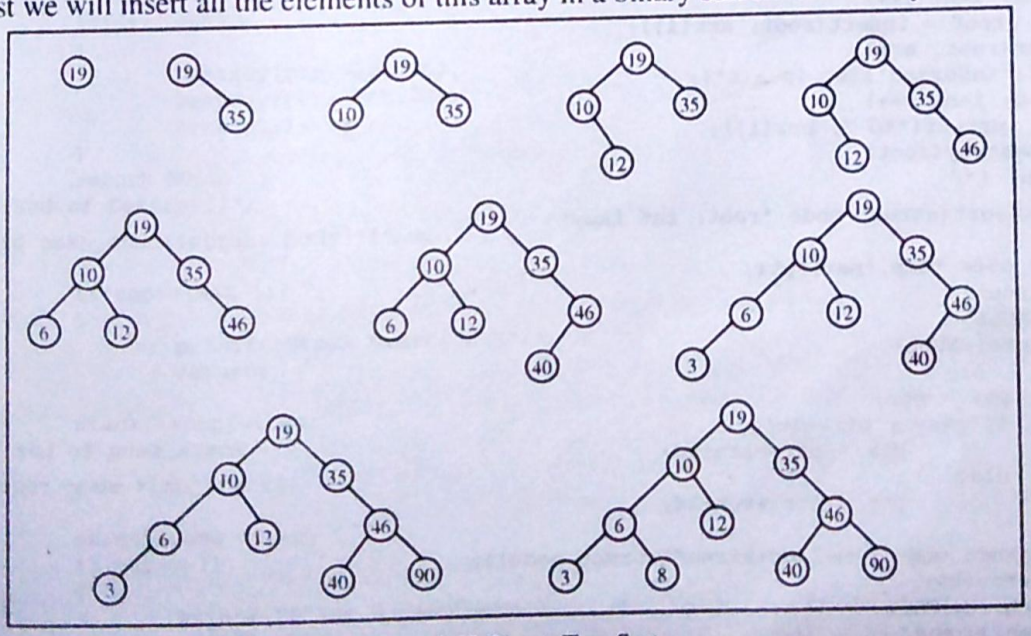


Figure 8.17 Binary Tree Sort

The inorder traversal of this binary search tree is-

3, 6, 8, 10, 12, 19, 35, 40, 46, 90

These elements are copied back to the array in this order and we get sorted array.

3	6	8	10	12	19	35	40	46	90
0	1	2	3	4	5	6	7	8	9

```
/*P8.10 Binary tree Sort*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
struct node
{
        struct node *lchild;
        int info;
        struct node *rchild;
};
struct node *stack[MAX];
int top=-1;
void push_stack(struct node *item);
struct node *pop_stack();
int stack_empty();
struct node *insert(struct node *ptr, int item);
void inorder(struct node *ptr, int arr[]);
struct node *Destroy(struct node *ptr);

main()
{
        struct node *root=NULL;
        int arr[MAX],n,i;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        for(i=0; i<n; i++)
                root = insert(root, arr[i]);
        inorder(root, arr);
        printf("\nSorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ", arr[i]);
        root=Destroy(root);
}/*End of main( )*/

struct node *insert(struct node *root, int ikey)
{
        struct node *tmp,*par,*ptr;
        ptr = root;
        par = NULL;
        while(ptr!=NULL)
        {
                par = ptr;
                if(ikey < ptr->info)
                        ptr = ptr->lchild;
                else
                        ptr = ptr->rchild;
        }
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=ikey;
        tmp->lchild=NULL;
        tmp->rchild=NULL;
        if(par==NULL)
                root=tmp;
        else if(ikey < par->info)
```

```
                par->lchild=tmp;
        else
                par->rchild=tmp;
        return root;
}/*End of insert( )*/
void inorder(struct node *root, int arr[])
{
        struct node *ptr=root;
        int i=0;
        if(ptr==NULL)
        {
                printf("Tree is empty\n");
                return;
        }
        while(1)
        {
                while(ptr->lchild!=NULL)
                {
                        push_stack(ptr);
                        ptr = ptr->lchild;
                }
                while(ptr->rchild==NULL)
                {
                        arr[i++]=ptr->info;
                        if(stack_empty())
                                return;
                        ptr = pop_stack();
                }
                arr[i++]=ptr->info;
                ptr = ptr->rchild;
        }
}/*End of inorder( )*/
/*Delete all nodes of the tree*/
struct node *Destroy(struct node *ptr)
{
        if(ptr!=NULL)
        {
                Destroy(ptr->lchild);
                Destroy(ptr->rchild);
                free(ptr);
        }
        return NULL;
}/*End of Destroy()*/
void push_stack(struct node *item)
{
        if(top==(MAX-1))
        {
                printf("Stack Overflow\n");
                return;
        }
        stack[++top]=item;
}/*End of push_stack()*/
struct node *pop_stack()
{
        struct node *item;
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        item=stack[top--];
```

```
        return item;
}/*End of pop_stack()*/
int stack_empty()
{
        if(top==-1)
                return 1;
        else
                return 0;
} /*End of stack_empty*/
```

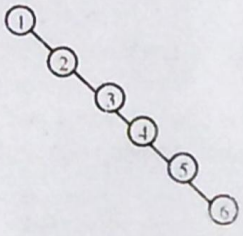
8.14.1 Analysis of Binary Tree Sort

We have seen that if a binary tree contains n nodes, then its maximum height possible is n and minimum height possible is $\lceil \log_2(n+1) \rceil$.

The maximum height n occurs when tree reduces to a chain or linear structure and in case of binary search tree this happens if the elements are inserted in ascending or descending order. The minimum height occurs when the tree is balanced.

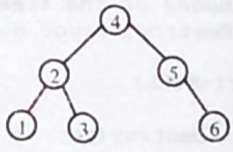
The main operation in binary tree sort is the insertion of elements in binary search tree. We have studied if h is the height of a binary search tree, then all the basic operations run in order O(h). Insertion of an element is also O(h) and so insertion of n elements will be O(nh). If the data that is to be sorted is in ascending or descending order then the tree that we get by inserting this data will have height n and so the insertion of n elements will be $O(n^2)$. If the data gives us an almost balanced tree then the insertion of n elements will be O(n log n).

Let us find out this order by counting the number of comparisons. We will take 6 numbers 1, 2, 3, 4, 5, 6 in sorted order, reverse sorted order and random order. The binary search trees obtained by inserting these numbers in different orders are -



(a) Data in sorted order
{ 1, 2, 3, 4, 5, 6}
Total comparisons =
0 + 2+ 3 +4 + 5+ 6 = 20

(b) Data in reverse sorted order
{ 6, 5, 4, 3, 2, 1}
Total Comparisons =
0 + 2+ 3 + 4 + 5+6 = 20



(c) Data in random order
{ 4, 2, 1, 5, 3, 6}
Total comparisons =
0 + 2 + 2 + 3 + 3 + 3 = 13

Figure 8.18

In case (a), no comparison is needed to insert node 1, only 2 comparisons are needed to insert node 2, and 3 comparisons are needed to insert node 3, and so on. Hence the total number of comparisons required to insert all the nodes is 20. Similarly in case (b) no comparison is needed to insert node 6, only 2 comparisons are needed to insert node 5, and 3 comparisons are needed to insert node 4, and so on. Here also the total number of comparisons required to insert all the nodes is 20. In case (c) no comparison is needed to insert node 4, and 2 comparisons each are required to insert nodes 2 and 5, and 3 comparisons each are required for inserting nodes 1, 3, 6 and so total number of comparisons in this case is only 13.

If the data is in sorted order or reverse sorted order then the total number of comparisons is given by-
$0 + 2 + 3 + 4 + 5 + \ldots\ldots\ldots\ldots + n = n(n+1)/2 \Rightarrow O(n^2)$

The main drawback of the binary tree sort is that if data is already in sorted order or in reverse order then the performance of binary tree sort is not good.

If data is in random order and suppose we get a balanced tree whose height is approximately log n then the number of comparisons can be given by-

$0 + 2 * 2^1 + 3 * 2^2 + 4 * 2^3 + \ldots\ldots\ldots (h) * 2^{h-1}$

This is because there can be maximum 2^L nodes at any level L, and L+1 comparisons are required to insert any node at level L as we have seen in case (c). The root node is at level 0 and the last level is h-1. The efficiency in this case is O(n log n).

So if the tree that we obtain is of height log n then the performance of binary tree sort is O(n log n). Generally with random data the chances of getting a balanced tree are good so we can say that average run time of binary tree sort is O(n log n).

After insertion of elements, some time is needed for traversal also. If we use a threaded tree then we can reduce this time by avoiding the use of stack. Binary tree sort is not an in-place sort since it requires additional O(n) space for the construction of tree. It is a stable sort.

8.15 Heap Sort

The heap tree that was introduced in the chapter on Trees has an important application in sorting. Heap sort is performed in two phases-

Phase 1 - Build a max heap from the given elements.
Phase 2 - Keep on deleting the root till there is only one element in the tree.

The root of a heap always has the largest element so by successively deleting the root, we get the elements in descending order i.e. first the largest element of list will be deleted then second largest and so on. We can store the deleted elements in a separate array or move them to the end of the same array that represents the heap.

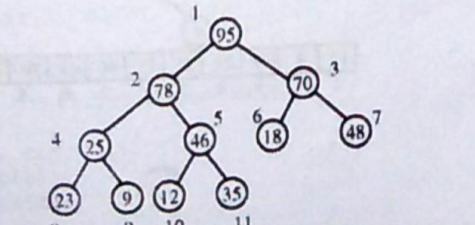
If we have `n` elements that are to be sorted, first we build a heap of size `n`. The elements `arr[1]`, `arr[2]`,.....`arr[n]` form the heap. Then root is deleted and we get a heap of size `n-1`. So now the elements `arr[1]`,`arr[2]`,.....`arr[n-1]` form the heap but `arr[n]` is not a part of the heap. The element deleted from root is the largest element, we can store it in `arr[n]` because `arr[n]` is not a part of heap now. Now the root from heap of size `n-1` is deleted and we get a heap of size `n-2`. The element deleted from root can be stored in `arr[n-1]`. This process goes on till we delete the root from heap of size 2 and get a heap of size 1 and this time the element deleted from root is stored in `arr[2]`. This way the array that represented the heap becomes sorted.

Let us take an array and sort it by applying heap tree sort.

arr

25	35	18	9	46	70	48	23	78	12	95
1	2	3	4	5	6	7	8	9	10	11

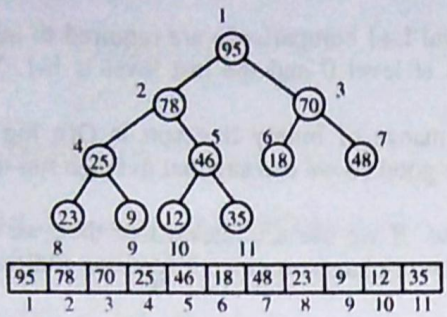
First this array is converted to a heap. The procedure of building a heap is described in chapter on trees. The heap obtained from this array is shown below-

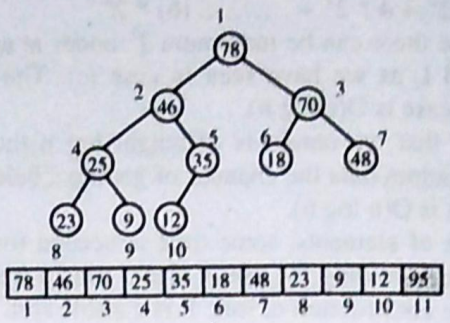


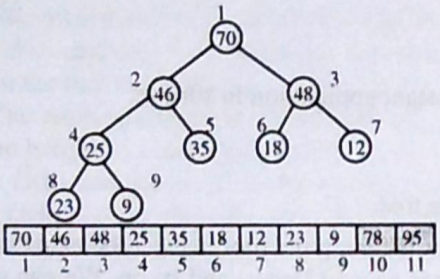
95	78	70	25	46	18	48	23	9	12	35
1	2	3	4	5	6	7	8	9	10	11

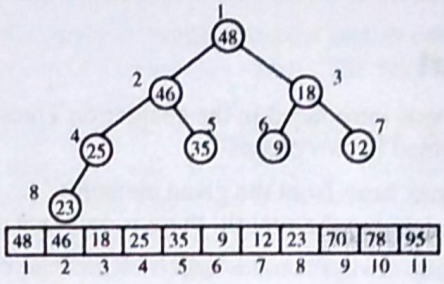
Figure 8.19

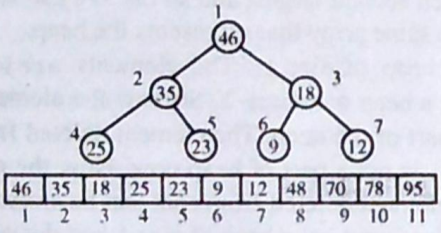
Now the root is repeatedly deleted from the heap. The procedure of deletion of root from the heap is described in chapter on trees.

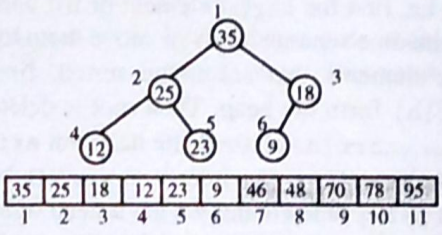


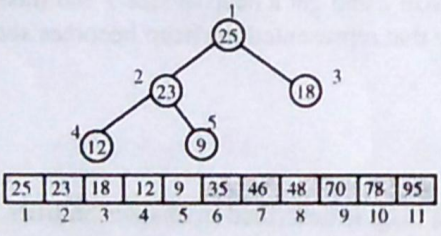


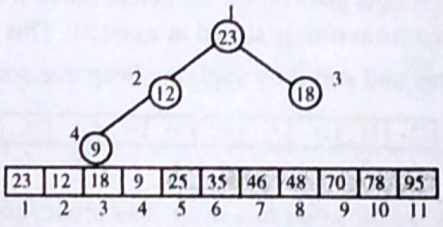


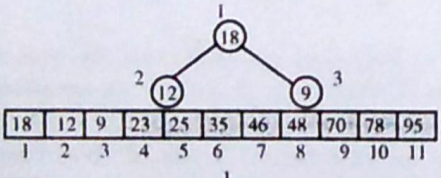


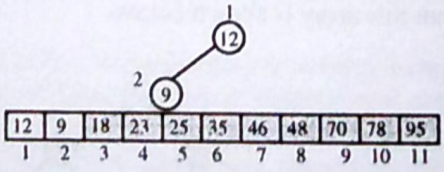


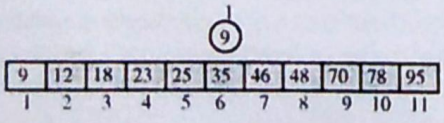












This way finally we see that the array `arr` becomes sorted. If we don't want to change the array that represents the heap we can take a separate array `s_arr[]` for the sorted output. The elements deleted from root can be stored in `s_arr[n]`, `s_srr[n-1]` and so on till `arr[1]`. This time we have to copy the element in position 1 also.

```
/*P8.11 Program of sorting through heapsort*/
```