Sorting Algorithms

The Sorting Problem

- **Input:**
  - A sequence of **n** numbers $a_1, a_2, \ldots, a_n$
- **Output:**
  - A permutation (reordering) $\mathbf{a_1', a_2', \ldots, a_n'}$ of input sequence such that $\mathbf{a_1' \leq a_2' \leq \cdots \leq a_n'}$

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Some Definitions

- **Internal Sort**
  - The data to be sorted is all stored in the computer's **main memory**.
- **External Sort**
  - Some of the data to be sorted might be stored in some **external, slower, device**.
- **In Place Sort**
  - The amount of **extra space** required to sort the data is **constant with the input size**.

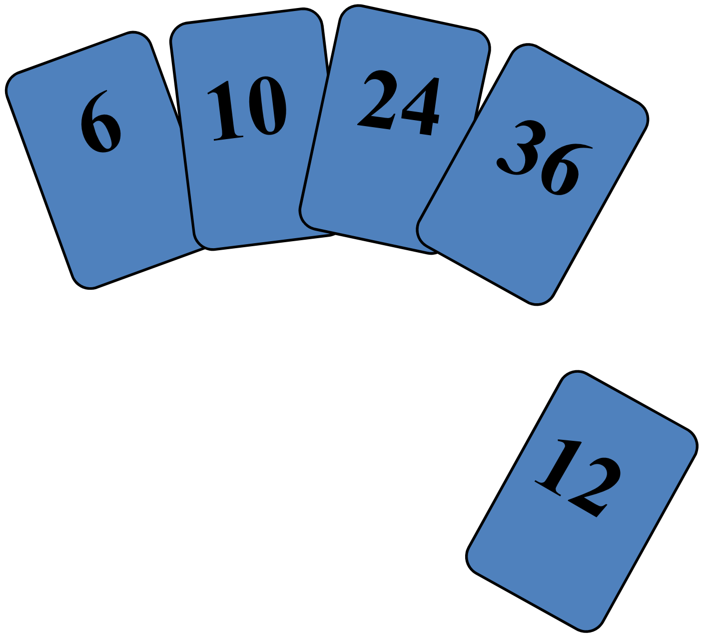
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion Sort

- **Idea: sorting a hand of playing cards**
  - Start with an empty left hand and the cards facing down on the table.
  - Remove **one card at a time** from the table, and insert it into the correct position in the left hand
    - **compare** it with each of the cards already in the hand, **from right to left**
  - The cards held in the left hand are sorted

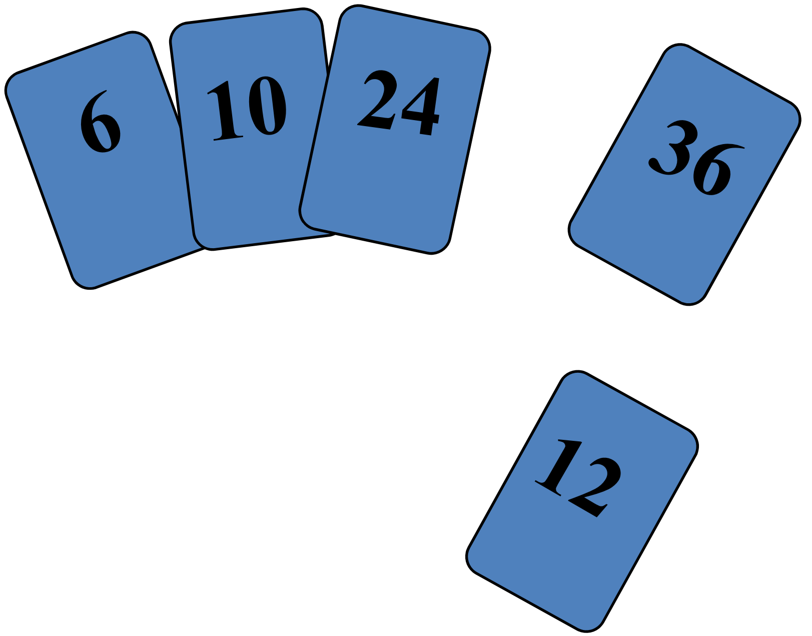
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion Sort



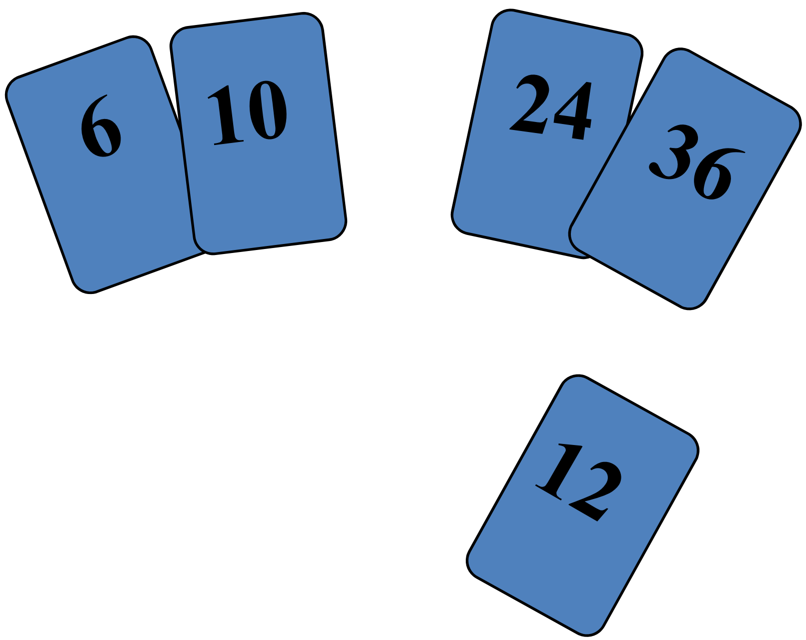
To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Insertion Sort



Insertion Sort

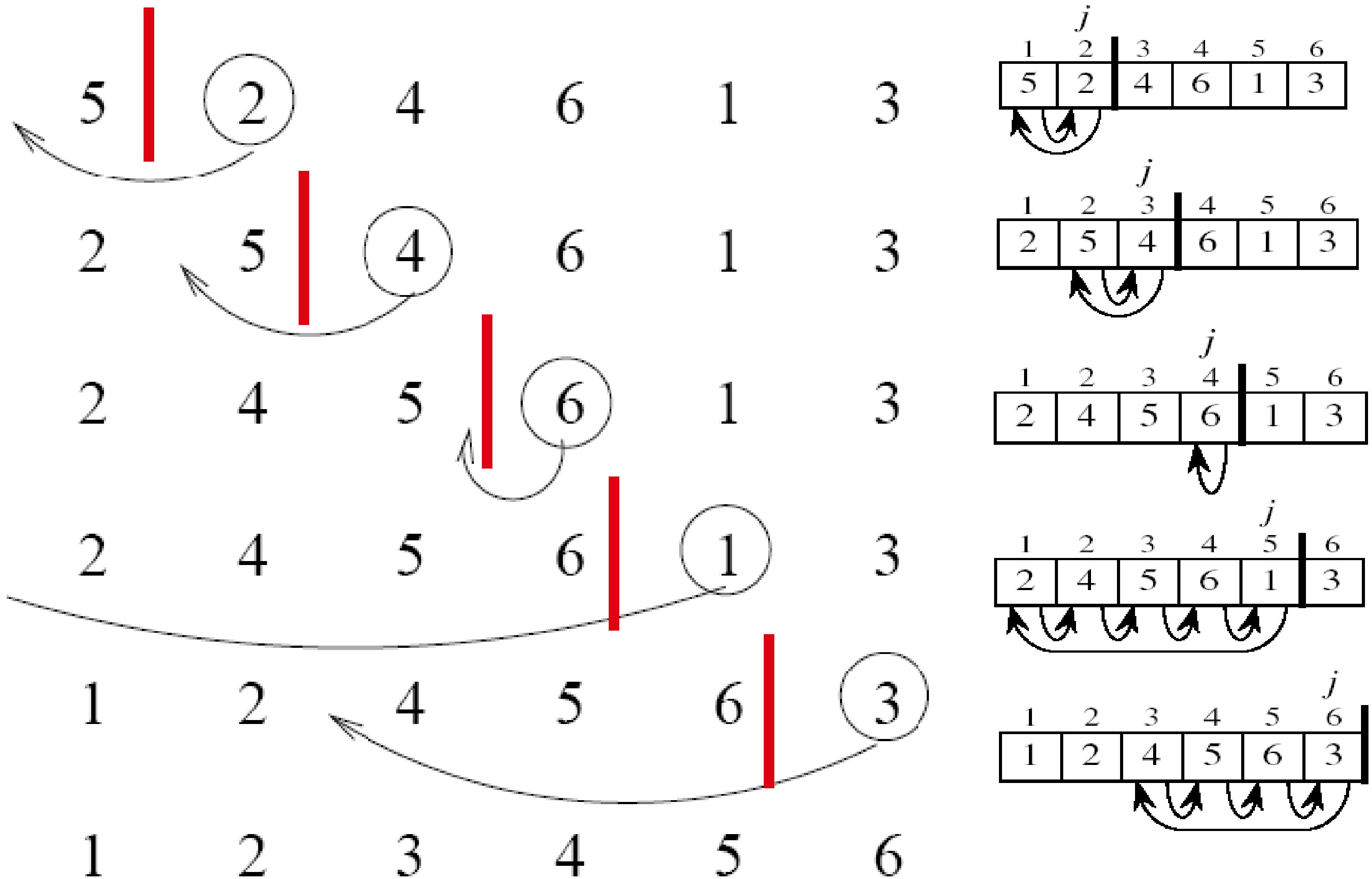
Sorts the elements in place

```
void insertionsort( int x[], int n )
{
    int temp, k, i;
    for( j = 2; j < = n; j ++)
    {
        temp = x[j];
        for( i = j-1; i >= 1 && temp < x[i]; i--)

                x[i+1]=x[i];
        x[i+1]=temp;
    }
}
```

temp ***holds the value to be inserted in the sorted array x [1..j-1]***

Shift the elements larger than temp to the right

Insertion Sort



Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Insertion Sort

- **Best case**
  - Array is already sorted
  - Outer loop executes n-1 times
  - Complexity $T(n)$ = **$O(n)$**

- **Worst case**
  - Array is in reverse order
  - $T(n) = 1 + 2 + \ldots. + (n-1) =$ **$O(n^2)$**

Insertion Sort

- **Average case**
  - The probability that $\mathbf{k^{th}}$ insertion requiring 1, 2, .., k number of comparisons is same = $\mathbf{1/k}$.
  - The expected number of comparisons for $\mathbf{k^{th}}$ insertion =1/k + 2/k +……….+ k/k = $\mathbf{(k+1)/2}$

$$T(n)=2/2 + 3/2 + 4/2 + \ldots\ldots\ldots + n/2$$

$$= \mathbf{O(n^2)}$$

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Selection Sort

- **Idea:**
  - Find the **smallest** element in the array
  - Swap it with the element in the **first position**
  - Find the **second smallest** element and **swap** it with the element in the **second position**
  - Continue until the array is sorted

Selection Sort

```
void  selectionsort ( int a [], int n )
{
     int i, j, index;
     for ( i = 0; i < n ; i ++)
     {
          index=i;
          for ( j = i +1 ; j < n ; j++)
          {
               if ( a[j] < a[index] )
                    index = j;
          }
          swap ( &a[i] , &a[index] );
     }
}
```

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8
1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

Selection Sort

- Irrespective of the order of the element, you have to find out the minimum element in every iteration.
- Number of comparisons required to find out the minimum element in i^{th} iteration is n-i.

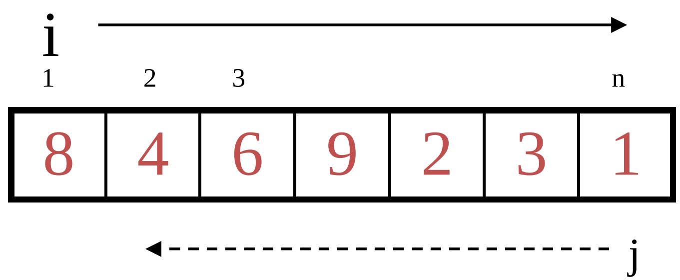
$$T(n)= (n-1) + (n-2) + \ldots\ldots\ldots\ldots + 3 + 2 + 1$$

$$=O(n^2)$$

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Bubble Sort

- **Idea:**
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



Bubble Sort

8	4	6	9	2	3	1

i = 1 j

8	4	6	9	2	1	3

i = 1 j

8	4	6	9	1	2	3

i = 1 j

8	4	6	1	9	2	3

i = 1 j

8	4	1	6	9	2	3

i = 1 j

8	1	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 1 j

1	8	4	6	9	2	3

i = 2 j

1	2	8	4	6	9	3

i = 3 j

1	2	3	8	4	6	9

i = 4 j

1	2	3	4	8	6	9

i = 5 j

1	2	3	4	6	8	9

i = 6 j

1	2	3	4	6	8	9

i = 7

j

Bubble Sort

```
Void bubblesort (int a [], int n)
{
    int i, j;
    for ( i = 1; i <= n; i++)
        for ( j = n; j > i; j --)
            if (a [j] < a [j-1])
                swap (&a [j], &a [j-1]);
}
```

Complexity = $O(n^2)$

Quick Sort

- Example of **Divide and Conquer** algorithm
- **Two phases**
  - Partition phase
    - Divides the work into half
  - Sort phase
    - Conquers the halves!

Quick Sort

- **Partition**
  - Choose a **pivot**
  - Find the position for the pivot so that
    - all elements to the **left are less**
    - all elements to the **right are greater**

< pivot	**pivot**	**> pivot**

Quick Sort

- **Conquer**
  - Apply the same algorithm to each half

< pivot				> pivot		
< p’	p’	> p’	pivot	< p”	p”	> p”

Quick Sort

```
quicksort( void *a, int low, int high )
{
    int pivot;
    if ( high > low )
    {
        pivot = partition( a, low, high );
        quicksort( a, low, pivot-1 );
        quicksort( a, pivot+1, high );
    }
}
```

Divide

Conquer

Quick Sort

```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = high;
    while ( left < right )
    {
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a, left, right);
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```


Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Quick Sort

```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = high;
    while ( left < right )
    {
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return r
}
```

Any item will do as the pivot, choose the leftmost one!

23	12	15	38	42	18	36	29	27

low high

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Quick Sort

```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = high;
    while ( left < right )
    {
        whi      eft] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( lef  < right ) SWAP(a, left, right);
    }
    a[l      = a[right];
    a[ri   t] = pivot_item;
    ret      ht;
}
```

Set left and right markers

left

right

23	12	15	38	42	18	36	29	27

low

pivot: 23

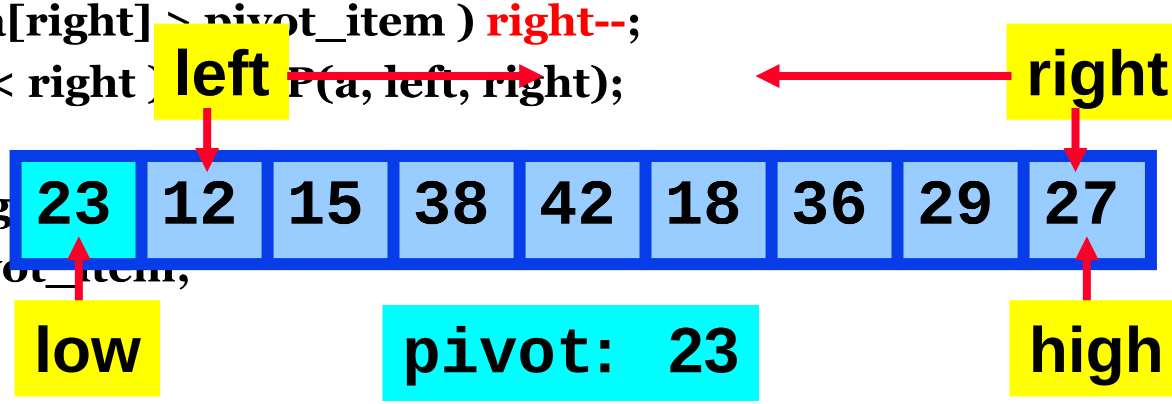
high

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Quick Sort

```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = high;
    while ( left < right )
    {
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) P(a, left, right);
    }
    a[low] = a[rig
    a[right] = pivot_item;
    return right;
}
```

Move the markers until they cross over



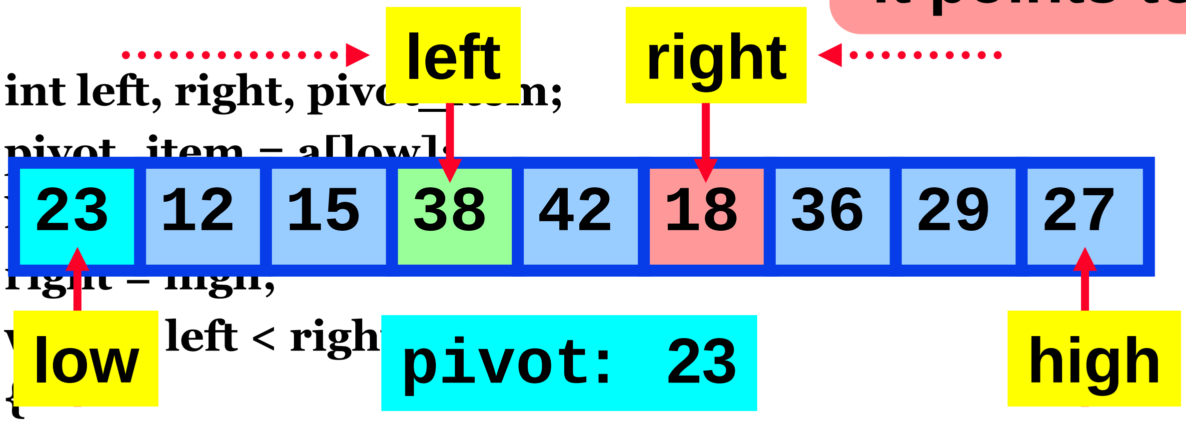
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Quick Sort

Move the left pointer while it points to items <= pivot

Move right similarly

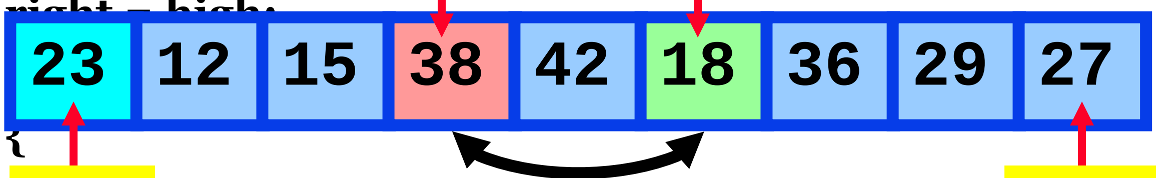
```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    right = high;
    while ( left < right )
    {
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a, left, right);
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```



Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Quick Sort

```
int partition( int *a, int low, int hig
{
    int left, right, pivot_item;
    pivot_item = a[lo
    left = low + 1;
    right = high;
    {
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a, left, right);
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```

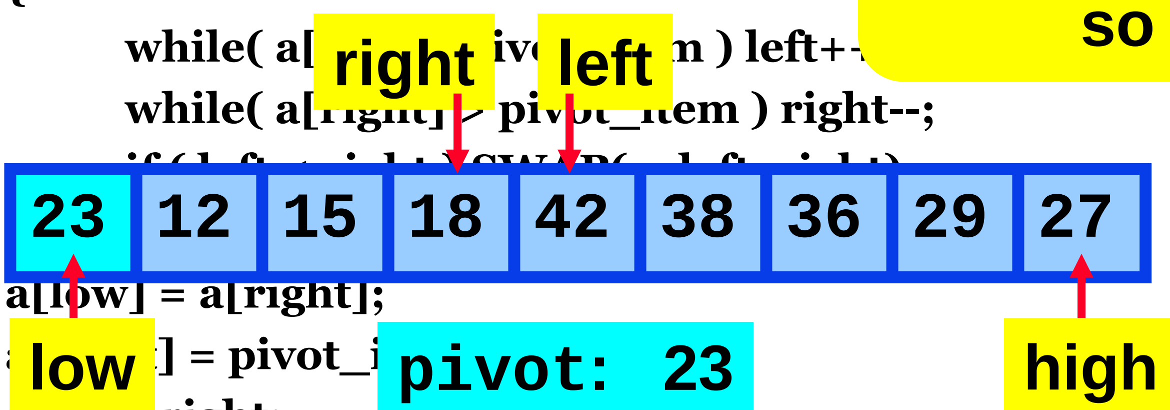


Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Quick Sort

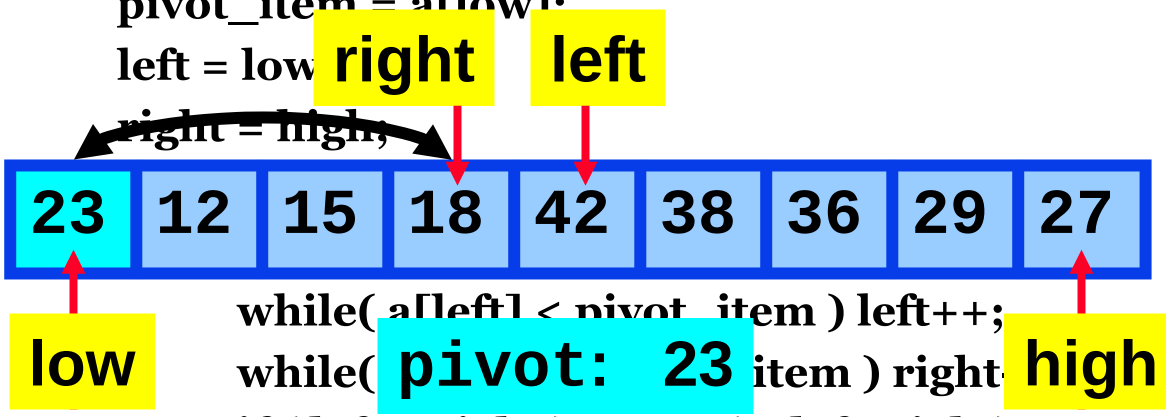
```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = high;
    while ( left < right )
    {
        while( a[left] <= pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a,left,right);
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```

`left` and `right` have swapped over, so stop



Quick Sort

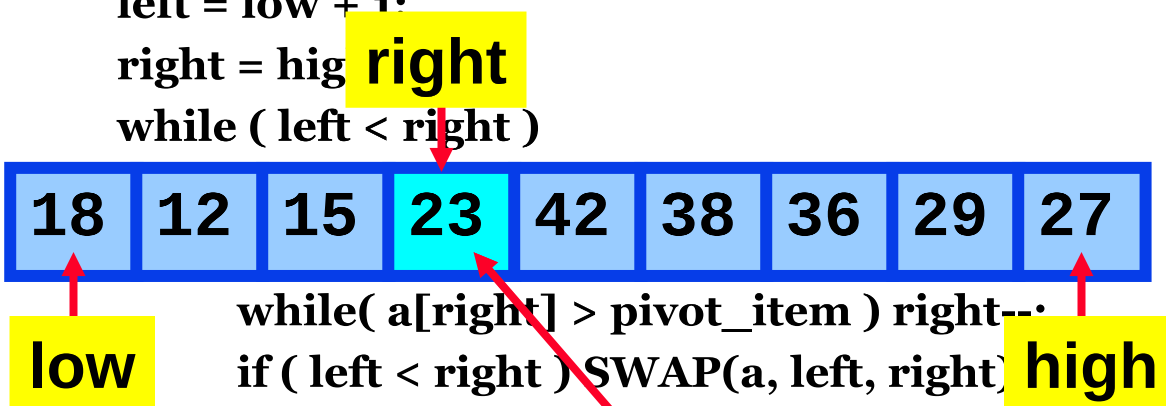
```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low;
    right = high;
        while( a[left] < pivot_item ) left++;
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a, left, right);
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```



Finally, swap the pivot and `right`

Quick Sort

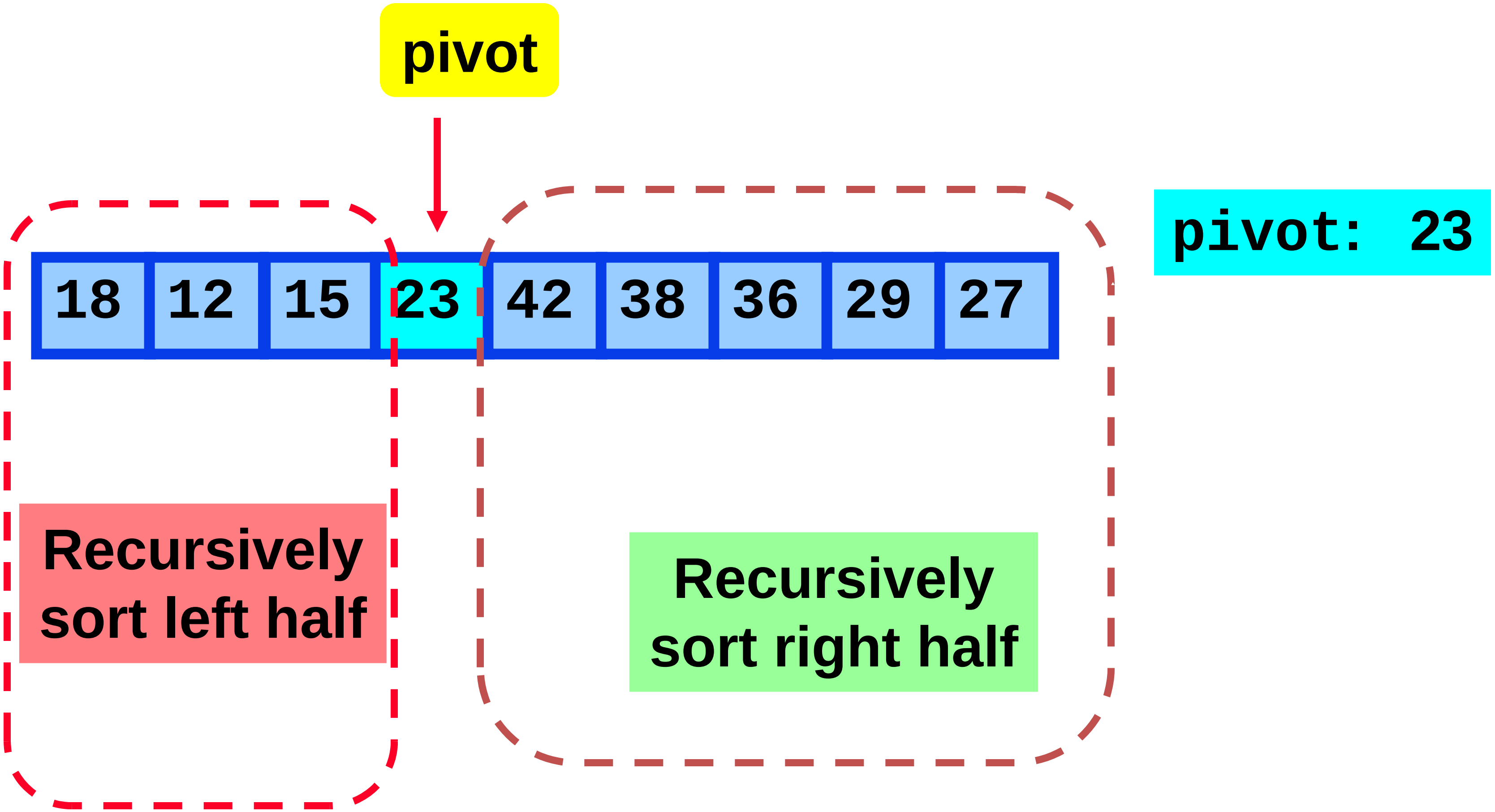
```
int partition( int *a, int low, int high )
{
    int left, right, pivot_item;
    pivot_item = a[low];
    left = low + 1;
    right = hig
    while ( left < right )

        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a, left, right)
    }
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```



pivot: 23

Return the position of the pivot

Quick Sort



Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

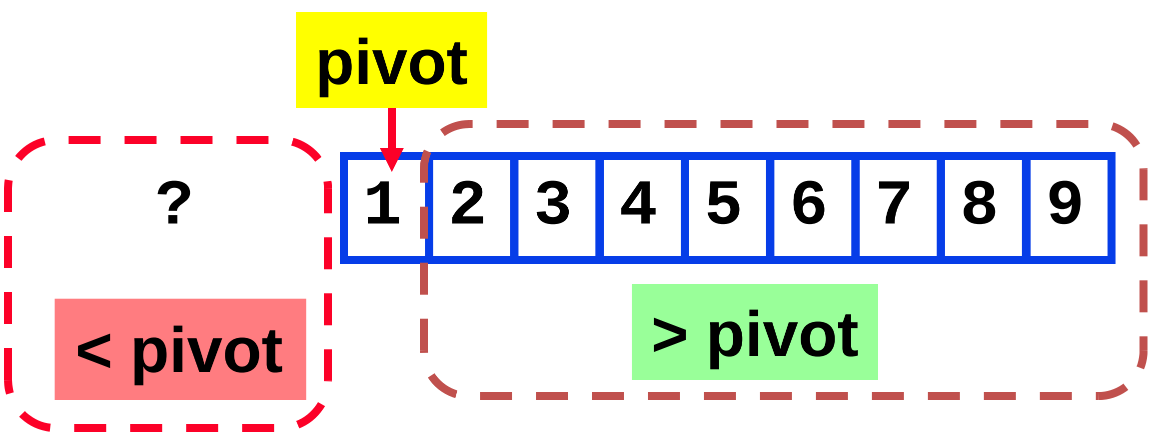
Quick Sort

- **Partition**
  - Check every item once $O(n)$
- **Conquer**
  - Divide data in half $O(\log_2 n)$
- **Total**
  - Product $O(n \log n)$
- ***But there's a catch ....***

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

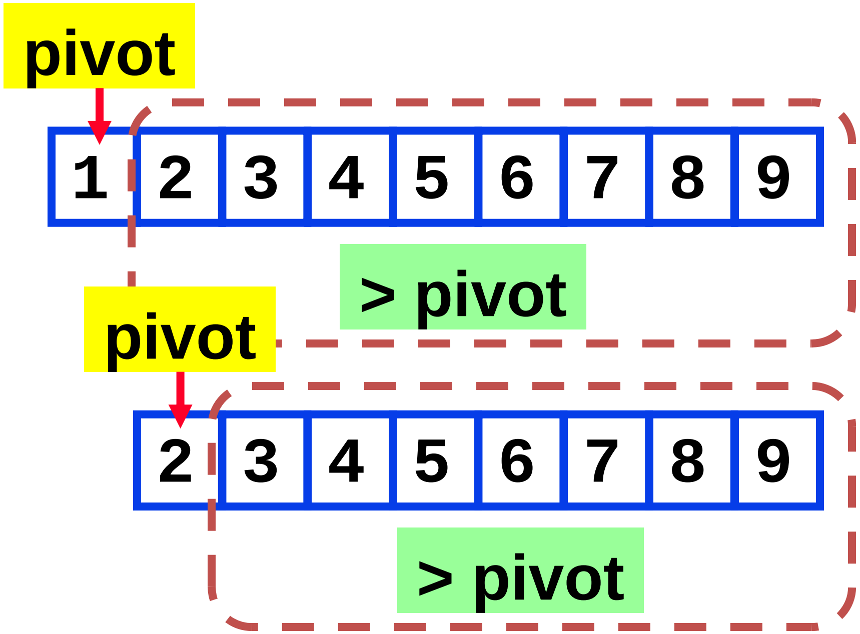
Quick Sort

- What happens if we use quicksort on data that's already sorted
  (or nearly sorted)
- ***We'd certainly expect it to perform well!!!!***



Quick Sort

- **Each partition produces**
  - a problem of size 0
  - and one of size *n*-1!
- **Number of partitions?**
  - *n* each needing time $O(n)$
  - Total $nO(n)$ or $O(n^2)$



? ***Quicksort is as bad as bubble or insertion sort***

Merge Sort

- **To sort an array A[p . . r]:**
  - **Divide**
    - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
  - **Conquer**
    - Sort the subsequences recursively using merge sort
      - When the size of the sequences is 1 there is **nothing more to do**
  - **Combine**
    - ***Merge*** **the two sorted subsequences**

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Merge Sort

```
void mergesort (int A [], int p, int r)
{
    int q;
    if ( p < r)
        {
            q = ( p + r ) / 2;        Divide
            mergesort ( A, p, q);
            mergesort (A, q+1, r);
            merge (A, p, q, r);       Combine
        }
}
```

Conquer

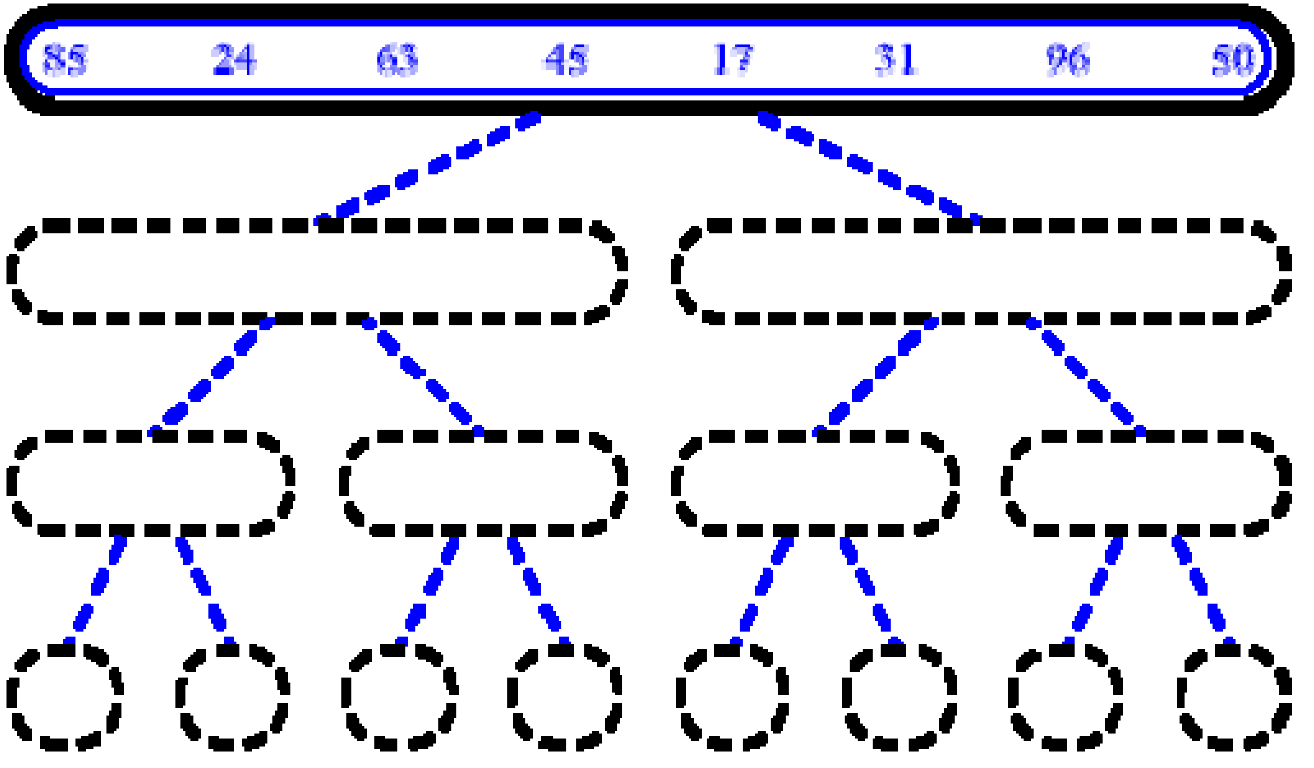
Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Merge Sort

```
void merge( int A [], int p, int q, int r)
{
    int B [100], i, j, k;
    i = p;    j=q+1;     k = 0;
    while ( i <= q &&   j <= r)
    {
        if ( A [i] < A [j])
                B [ k ++] = A [ i ++];
        else
                B [ k ++] = A [ j ++];
    }
    while ( i <= q)
        B [ k ++] = A [ i ++];
    while ( j <= r)
        B [ k ++] = A [ j ++];
}
```

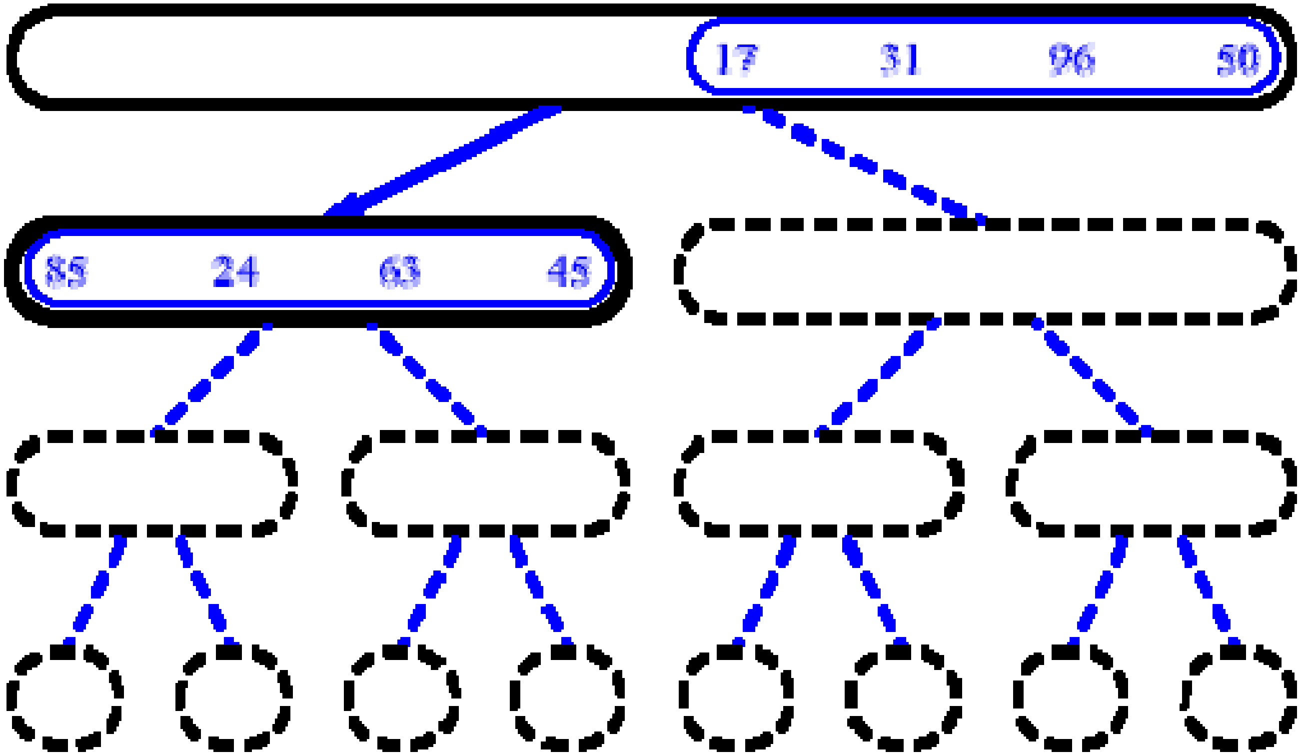

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Merge Sort

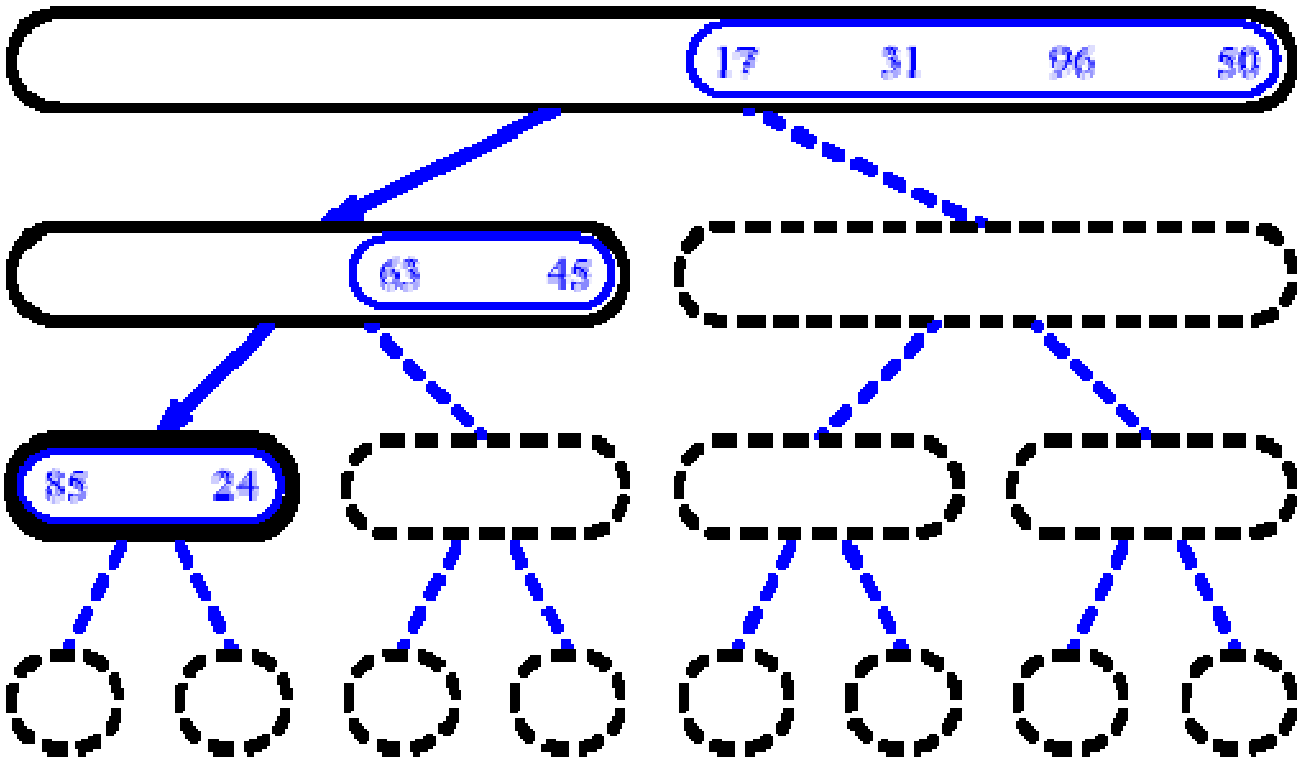


Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Merge Sort

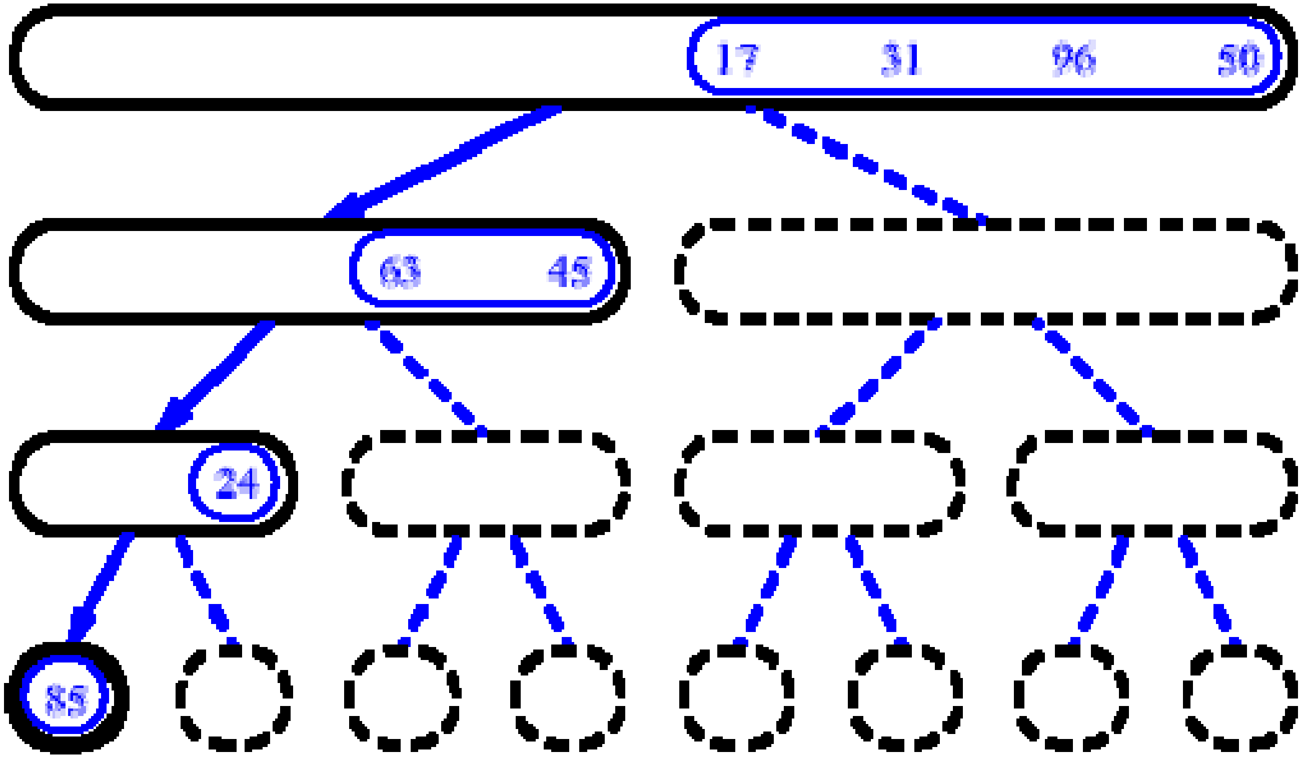


Merge Sort



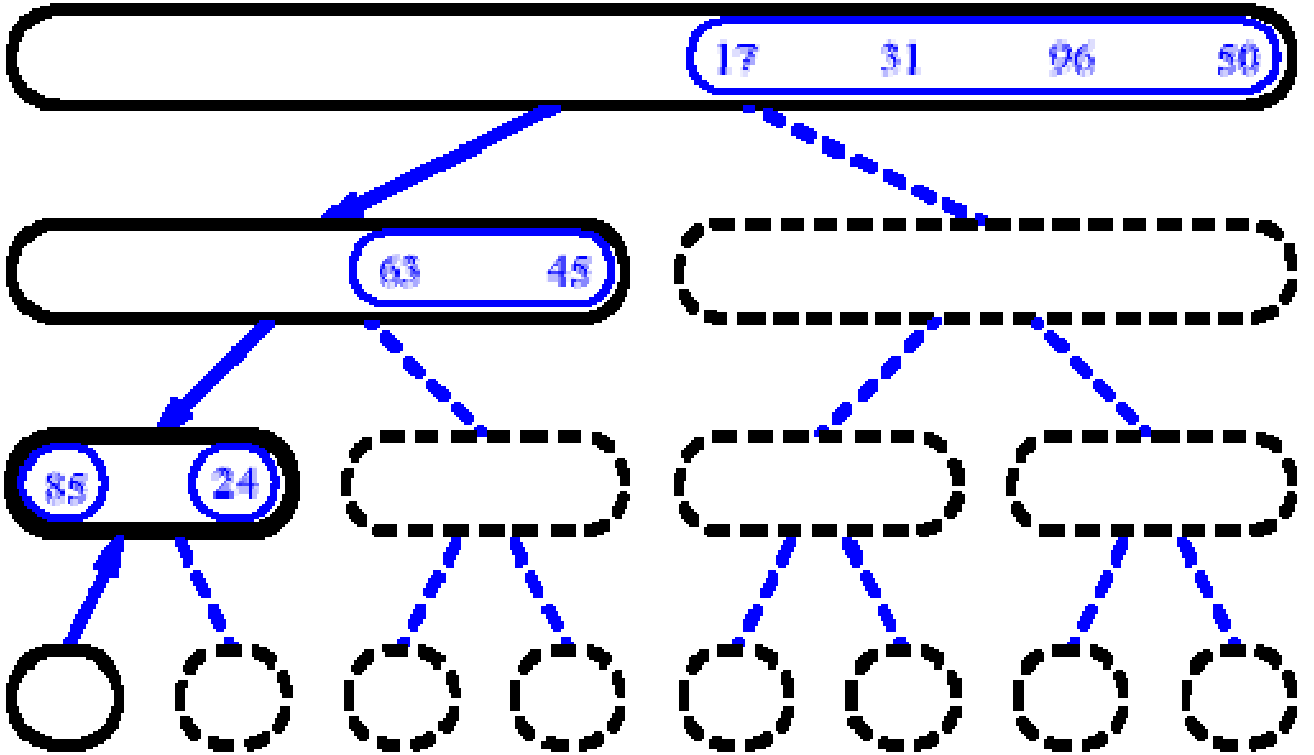
Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort

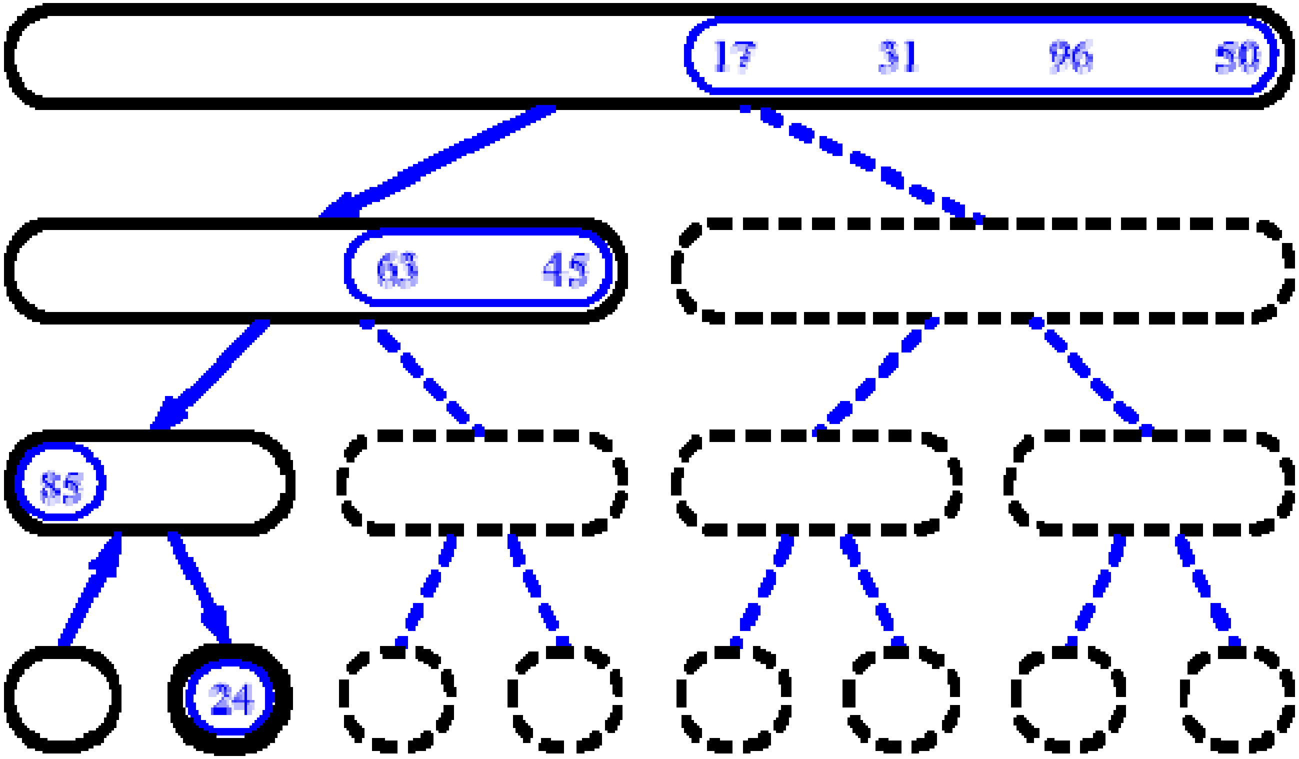


Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

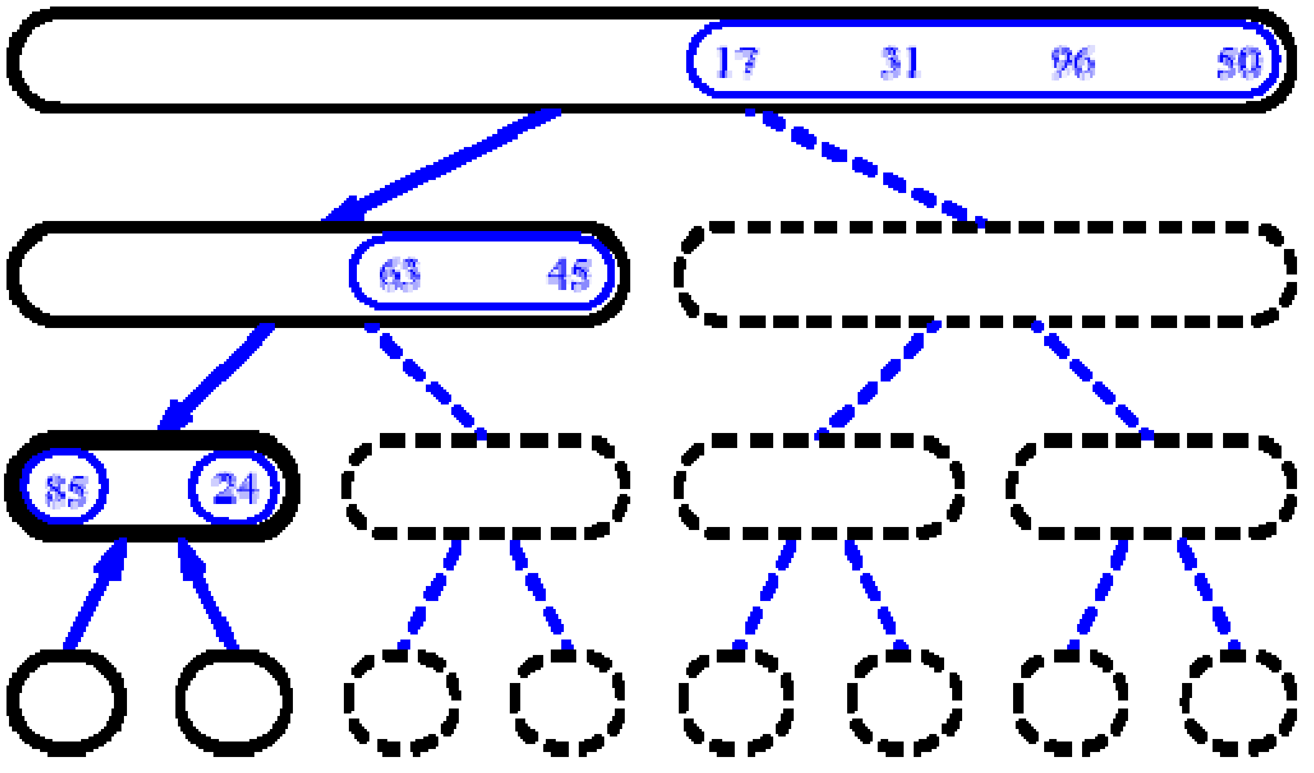
Merge Sort



Merge Sort

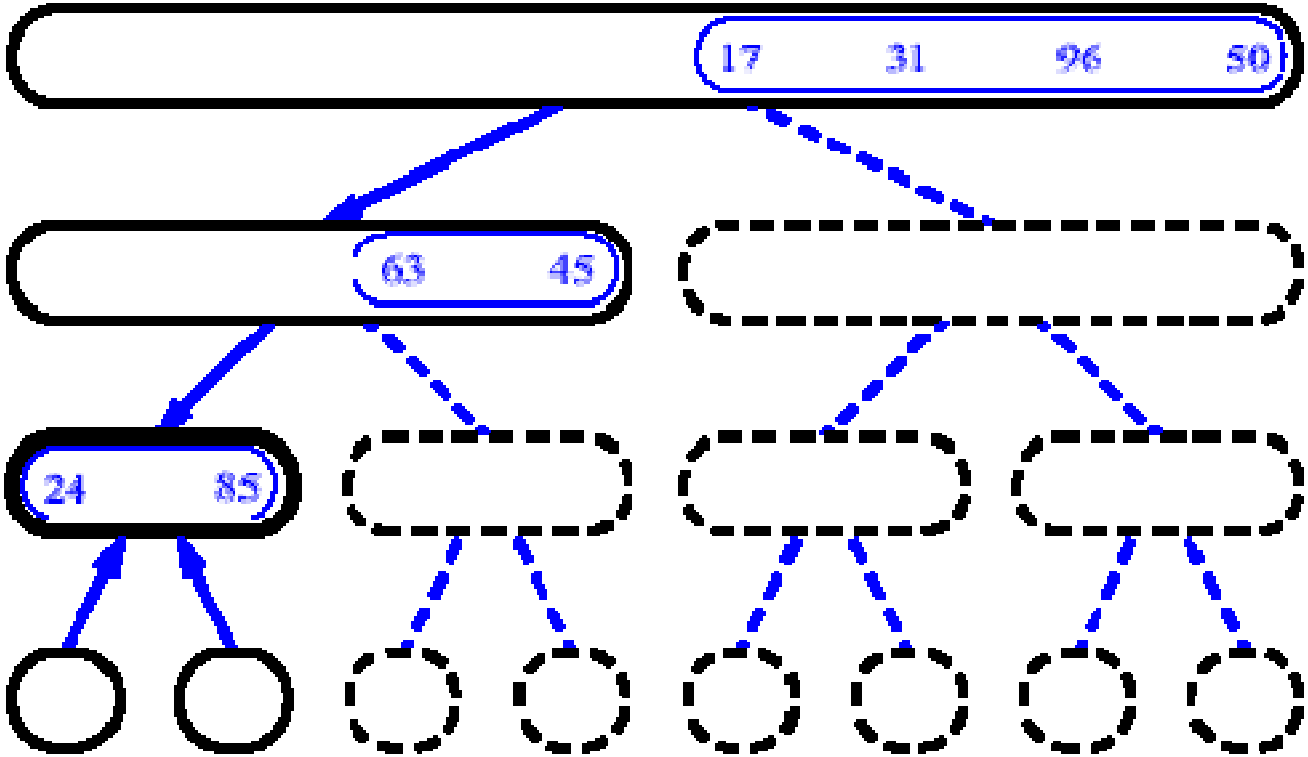


Merge Sort



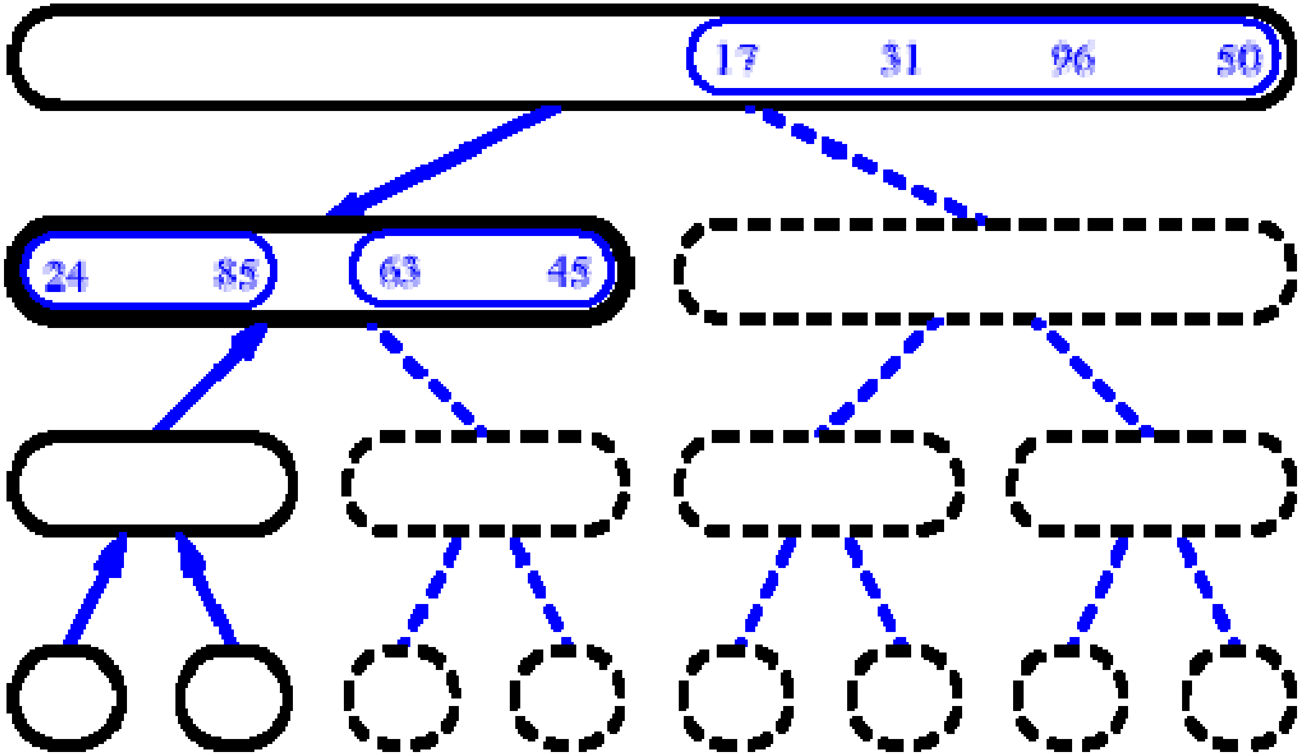
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Merge Sort



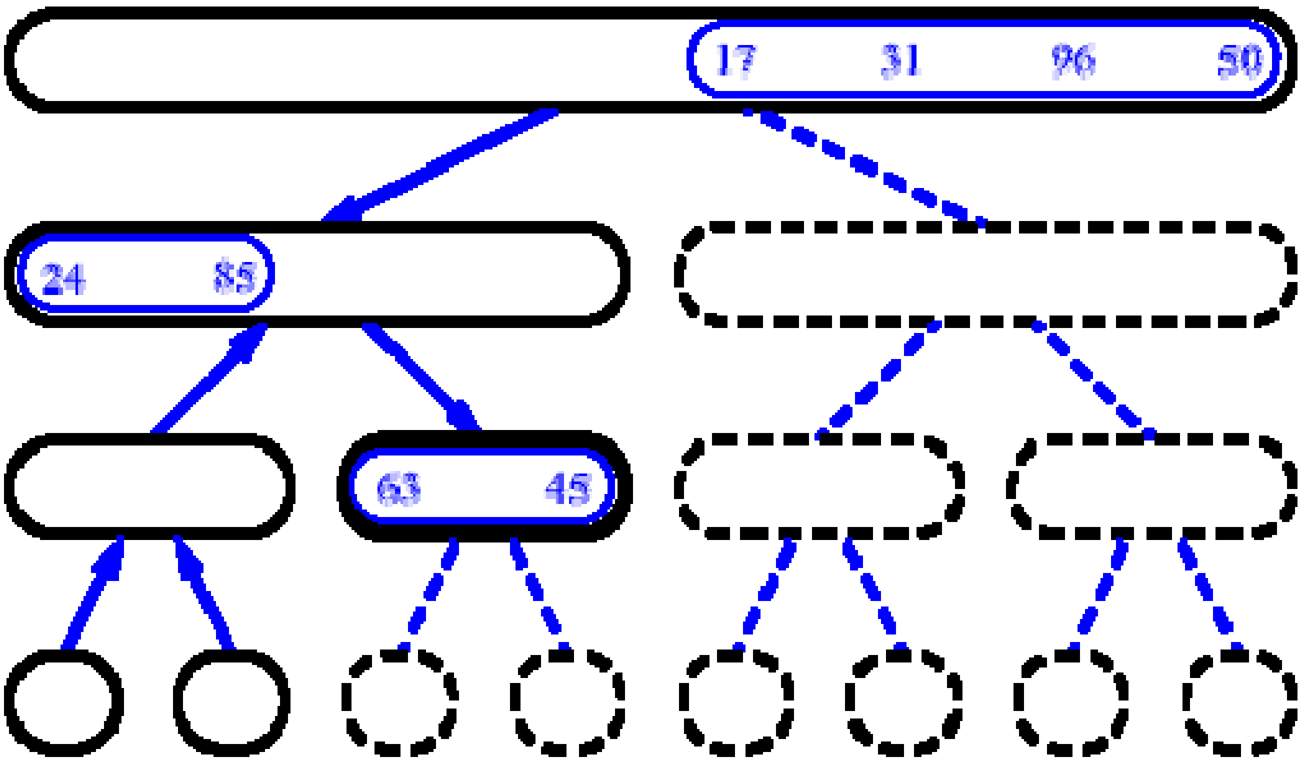
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Merge Sort



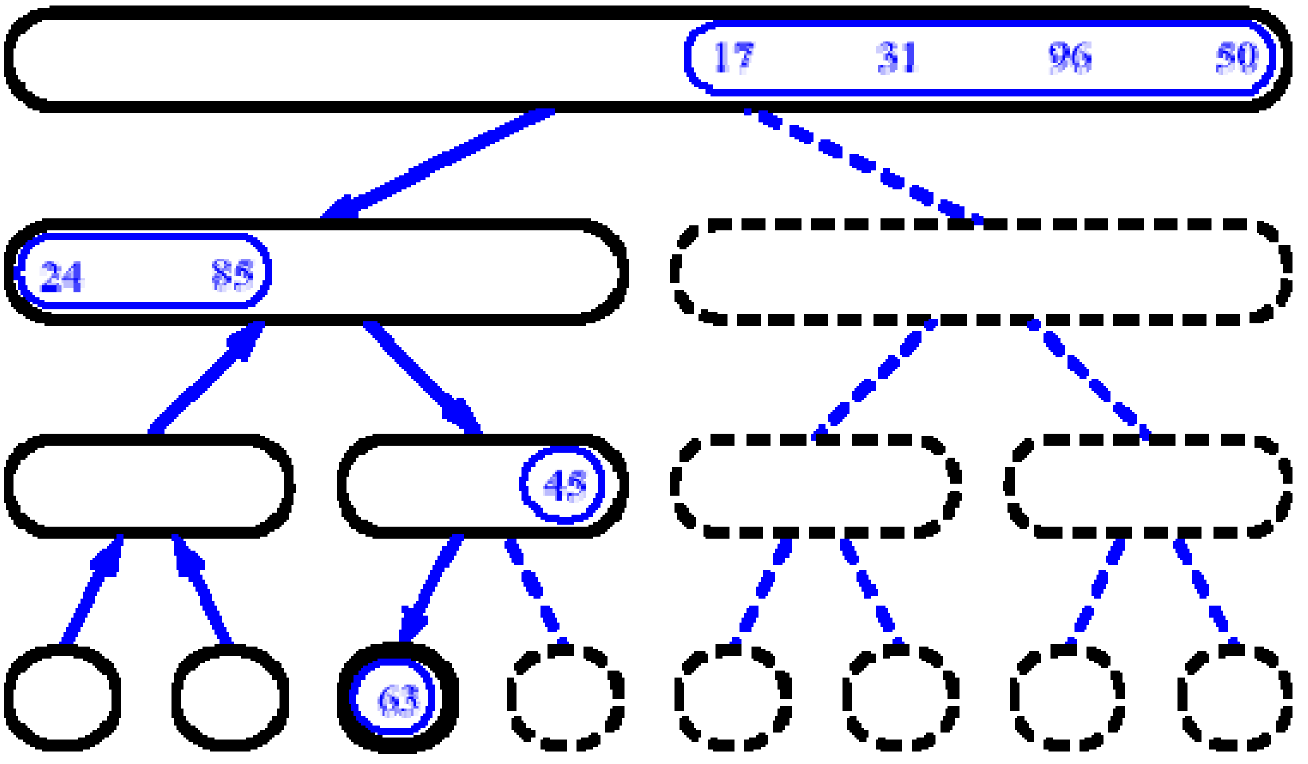
Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort



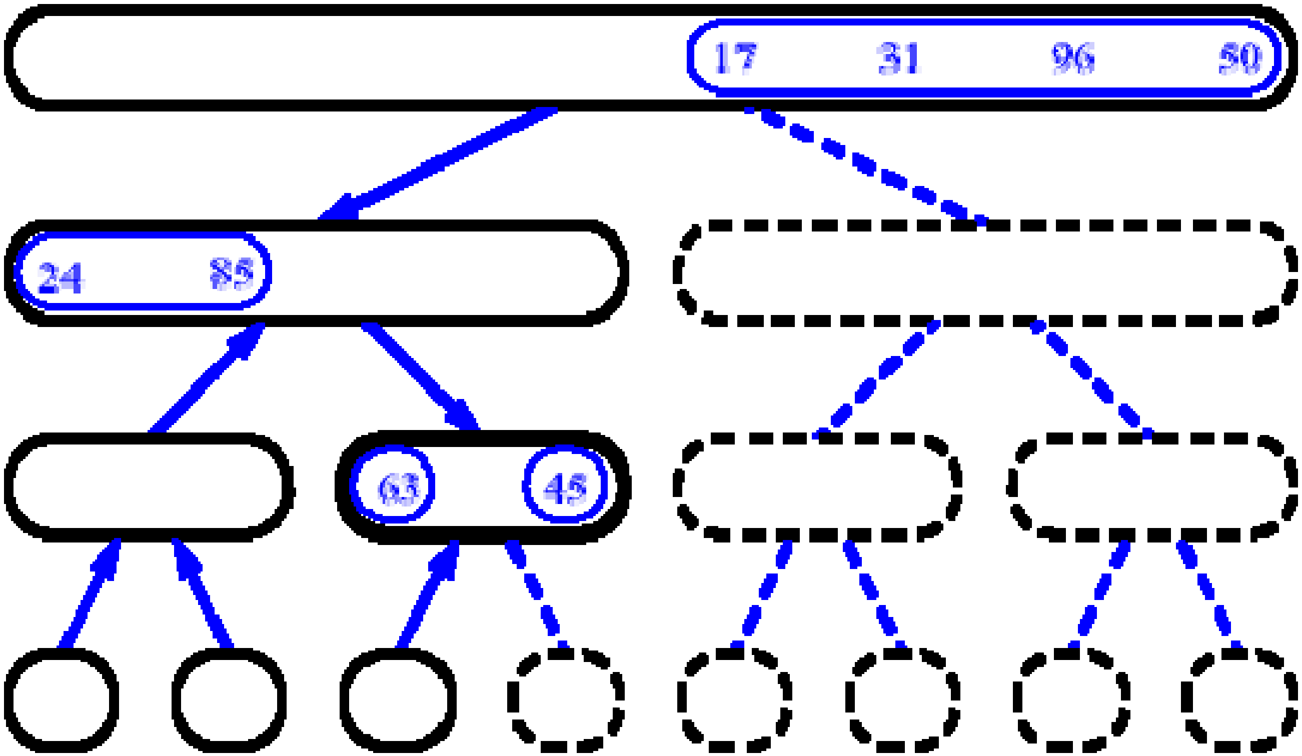
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Merge Sort



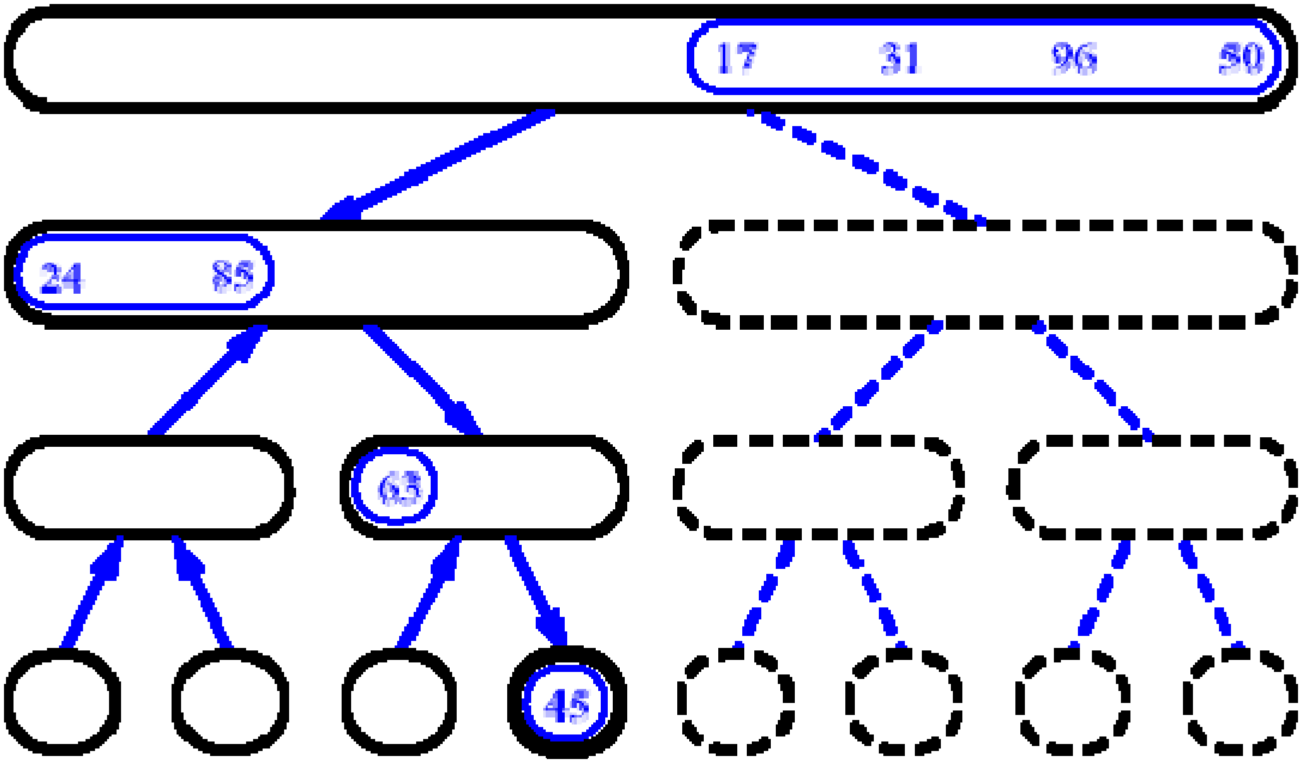
Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort

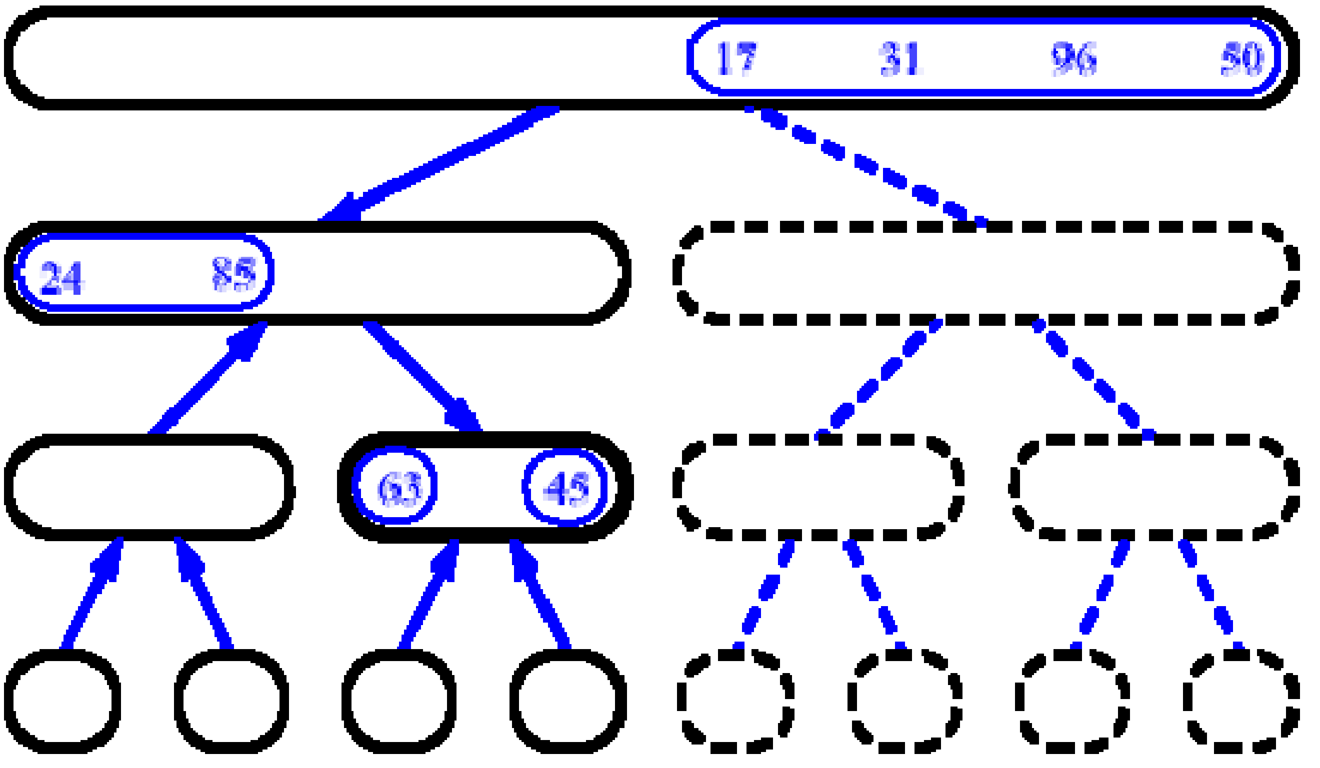


Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

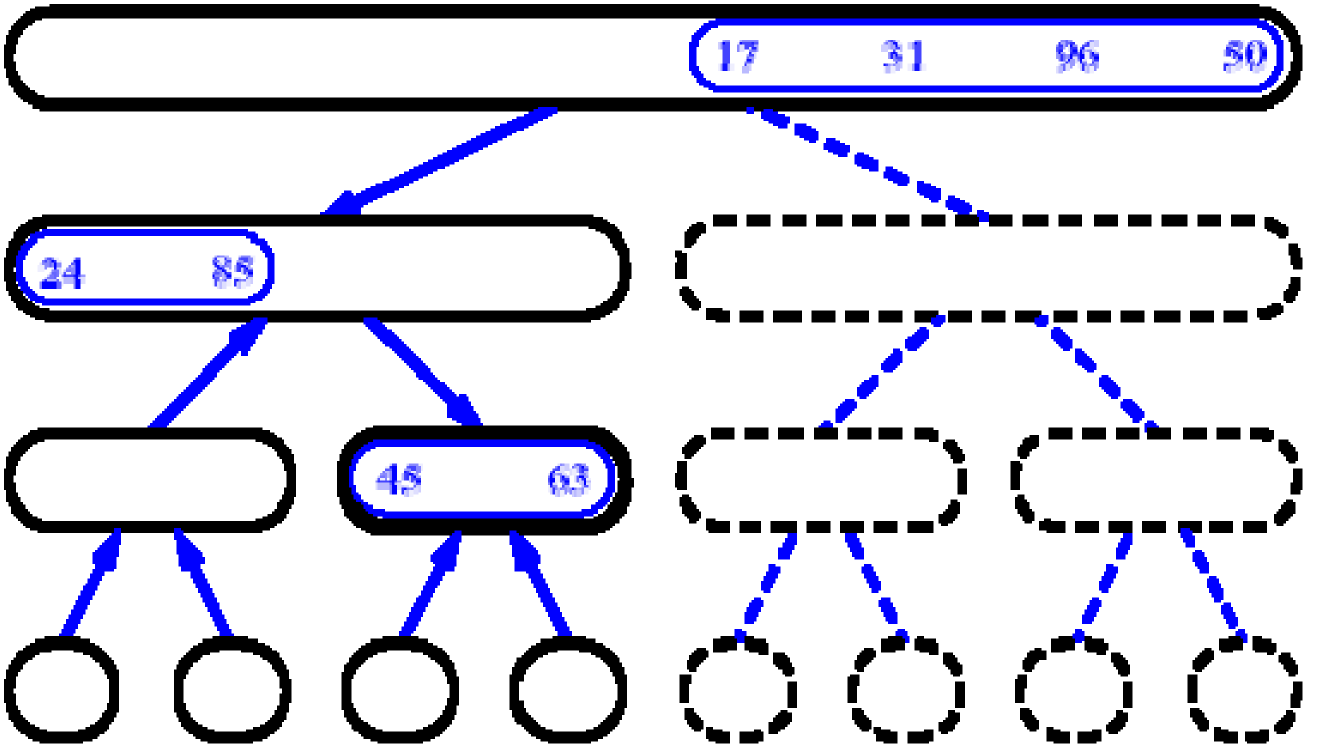
Merge Sort



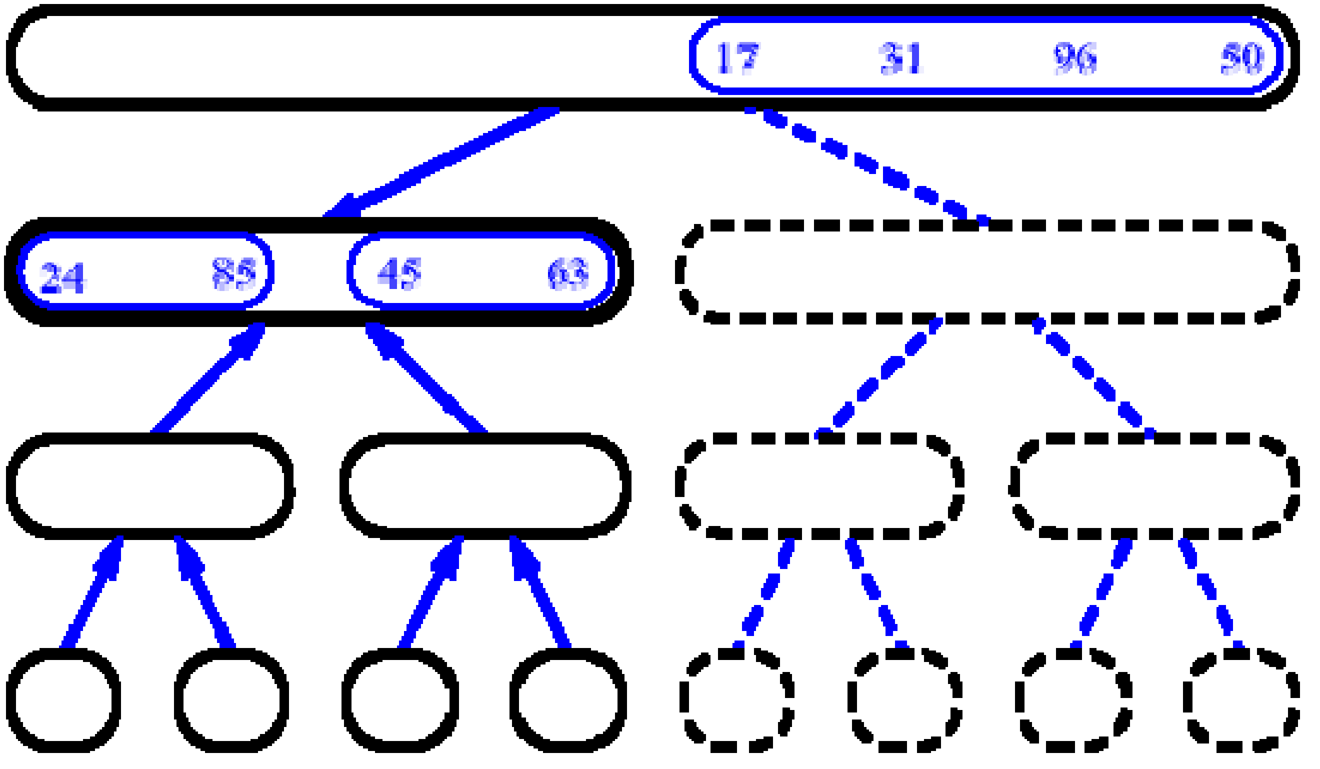
Merge Sort



Merge Sort

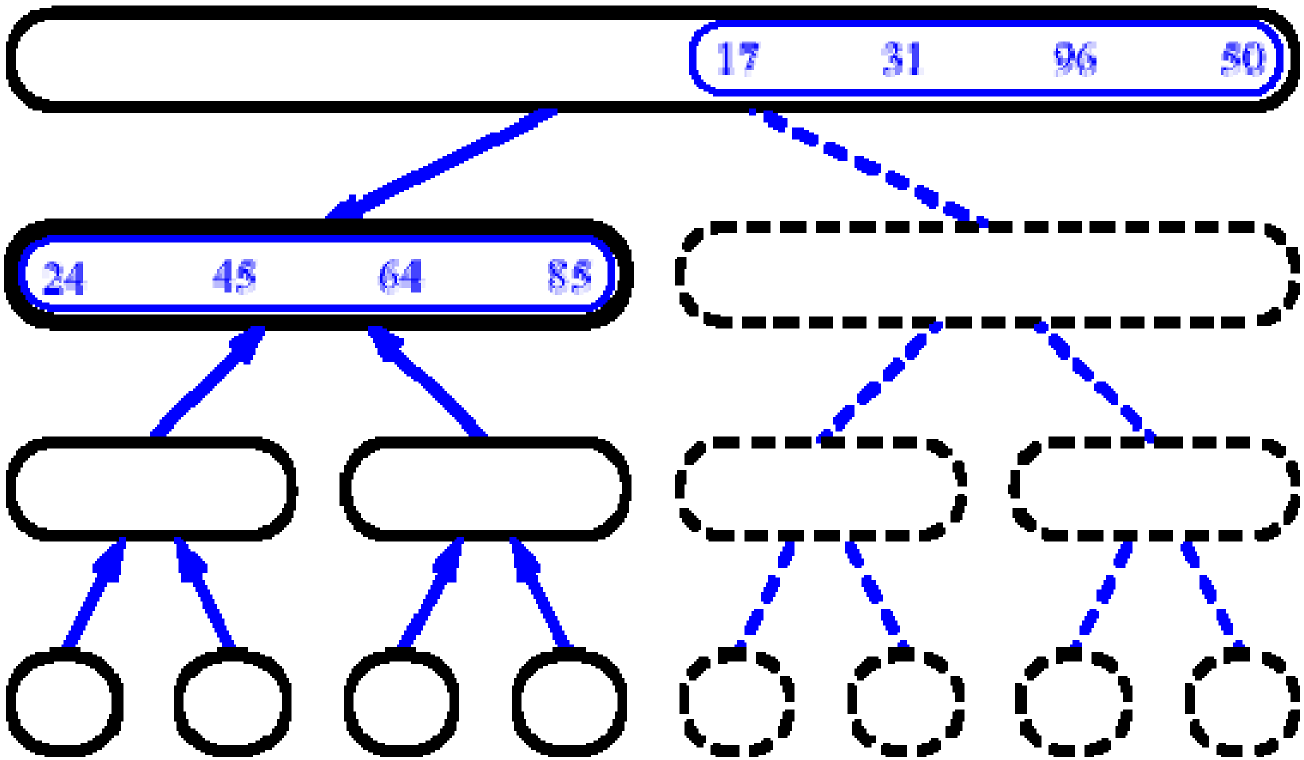


Merge Sort

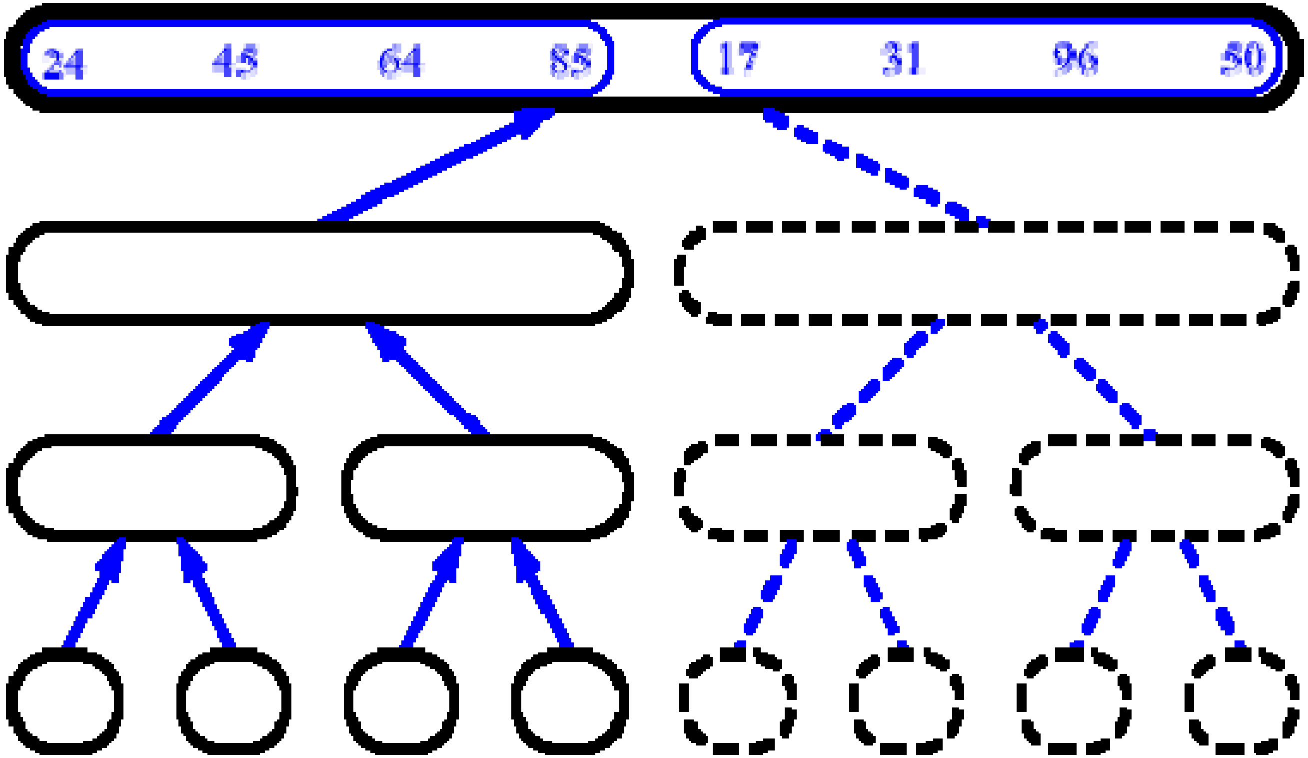


Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Merge Sort

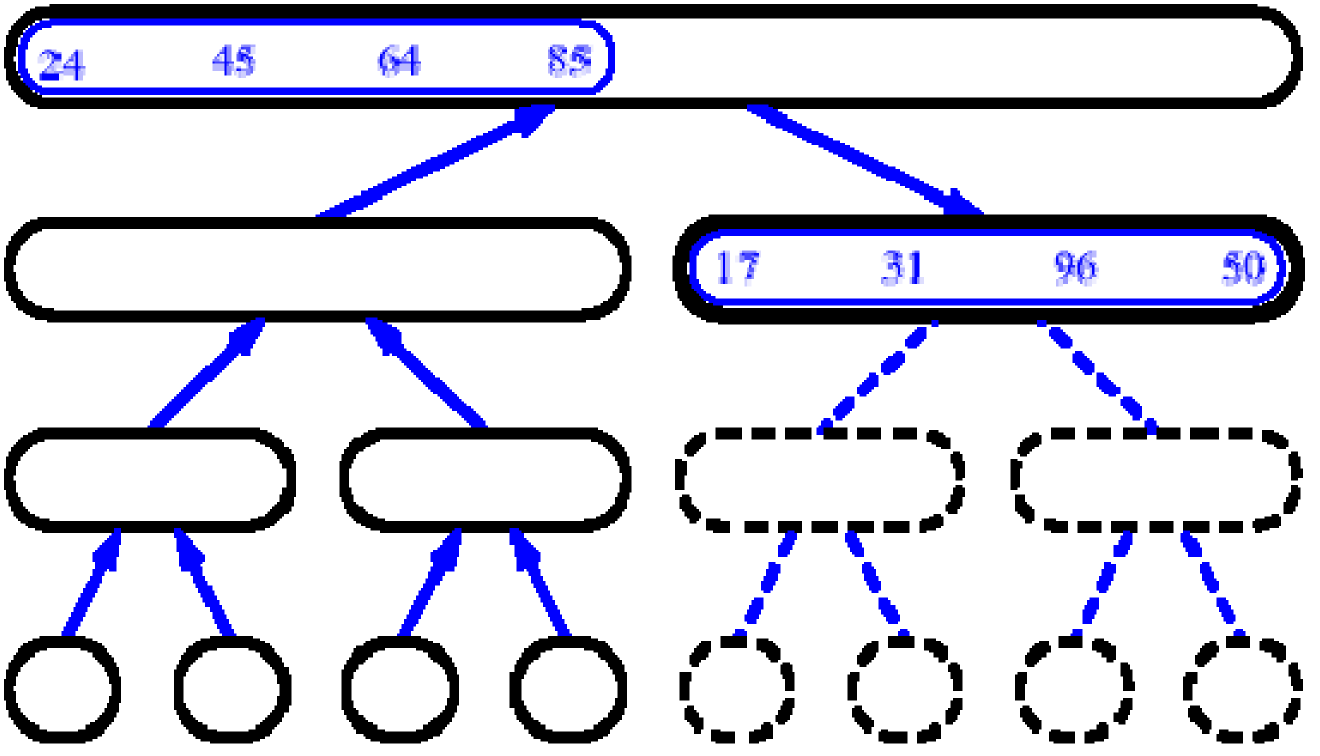


Merge Sort



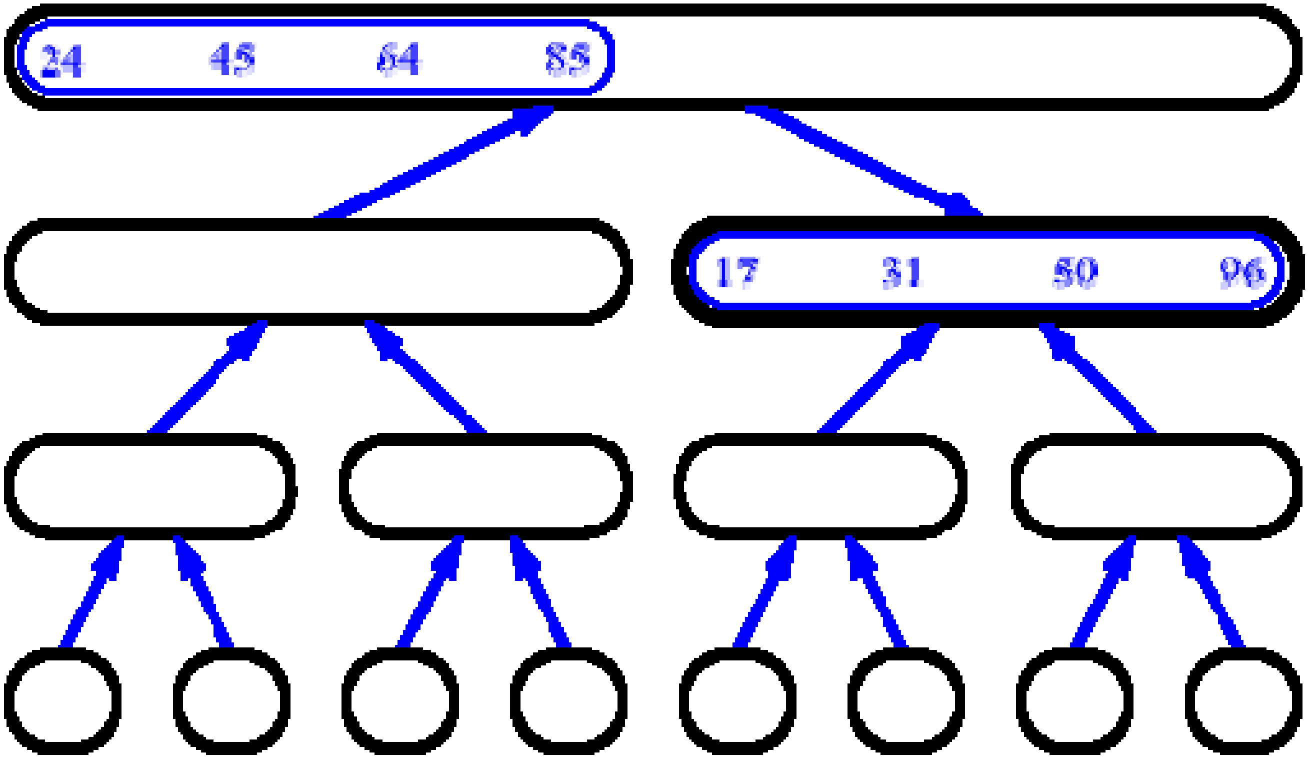
Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort



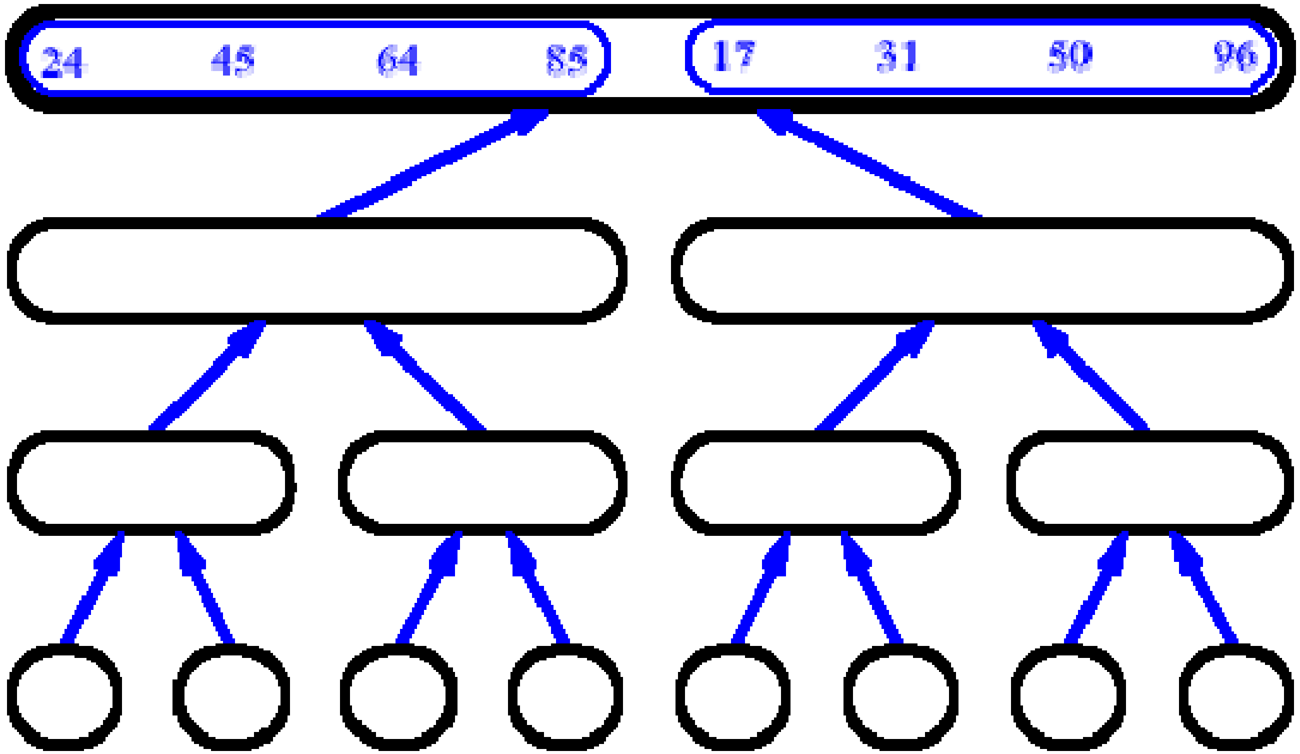
Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Merge Sort



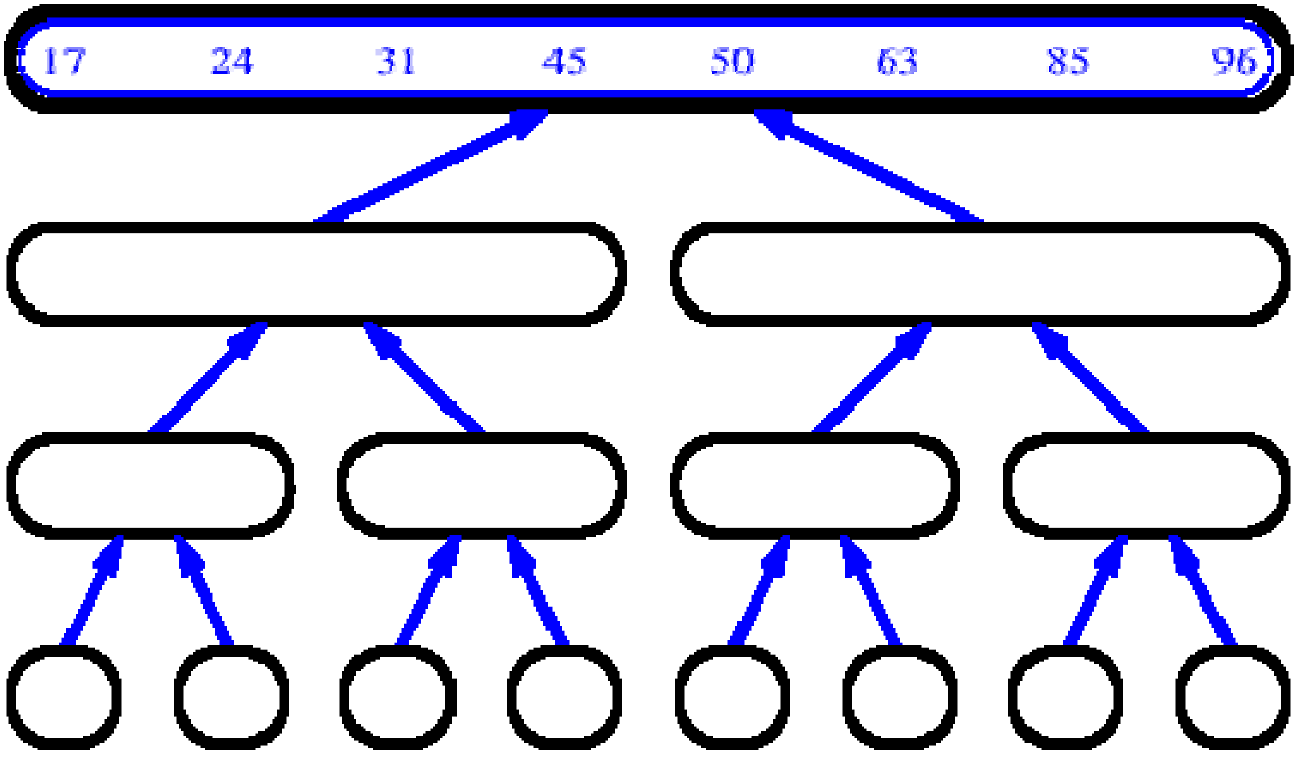
Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort



Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Merge Sort



Merge Sort

- **Running time $T\ (n)$ of Merge Sort:**
  - **Divide:** computing the middle takes **O(1)**
  - **Conquer:** solving 2 sub-problems takes $\mathbf{2T(n/2)}$
  - **Combine:** merging n elements takes $\mathbf{O(n)}$
  - **Total:**

$$T(n) = 2T(n/2) + O(n)$$

$$\Rightarrow T(n) = O(n \log n)$$

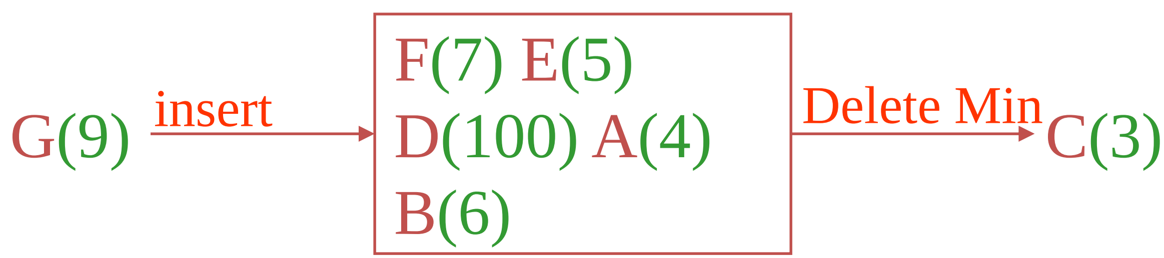
Priority Queues and Heaps

- **Consider applications**
  - **Ordering** CPU jobs
  - Printing **Jobs**
  - **Emergency** room admission processing

- **Problems?**
  - short jobs **should go first**
  - Hold jobs for a printer in **order of length**
  - most urgent cases **should go first**

Priority Queues and Heaps

- **Priority Queue property:**
  - For two elements in the queue, ***x*** and ***y***, if ***x*** has a lower ***priority value*** than ***y***, **x** will be ***deleted before y***



Priority Queues and Heaps

- **Priority Queue operations**
  - **Create**
  - **Destroy**
  - **Insert**
  - **Delete Min / Delete Max**
  - **Is_empty**

Priority Queues and Heaps

- **Unordered linked list**
  - **Insert : O(1)**
  - **Delete Min/Max : O(n)**
- **Ordered linked list**
  - **Insert : O(n)**
  - **Delete Min/Max : O(1)**
- **Ordered array**
  - **Insert : O(n)**
  - **Delete Min/Max : O(1)**
- **Balanced BST**
  - **Insert : O(log n)**
  - **Delete Min/Max : O(log n)**

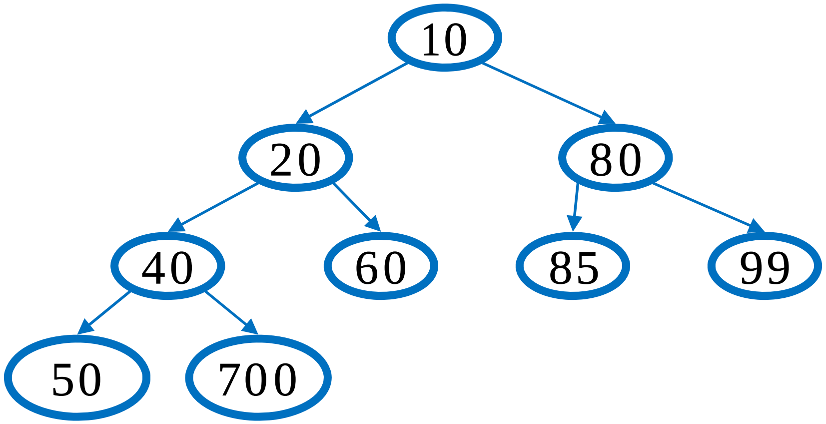
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Priority Queues and Heaps

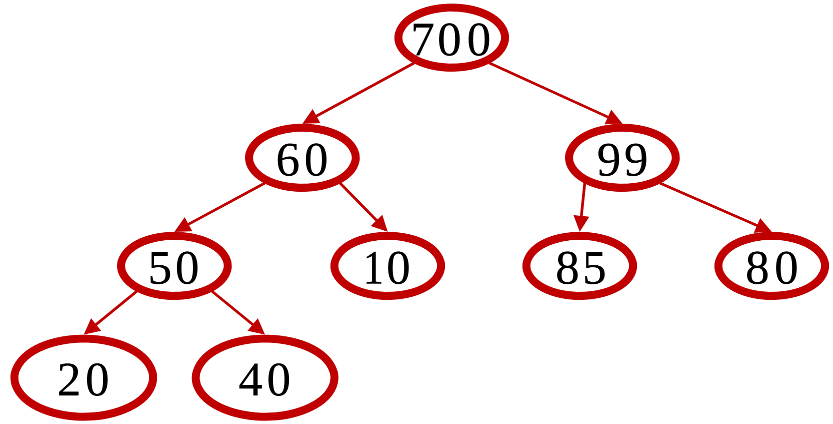
- A **heap** is a binary tree with two properties:
  - **Structure property**
    - A complete binary tree
      - Height of a complete binary tree with n elements is log n
  - **Heap-order property**
    - Parent's key is smaller (greater) than children's keys in Min Heap (Max Heap)
    - Result: Minimum (maximum) is always at the root

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

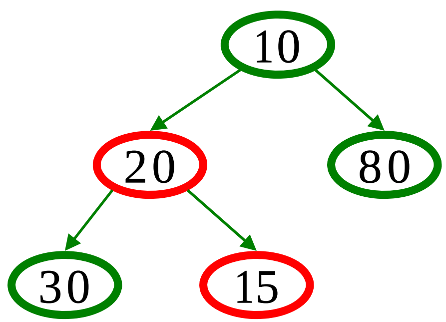
Priority Queues and Heaps



Min Heap



Max Heap

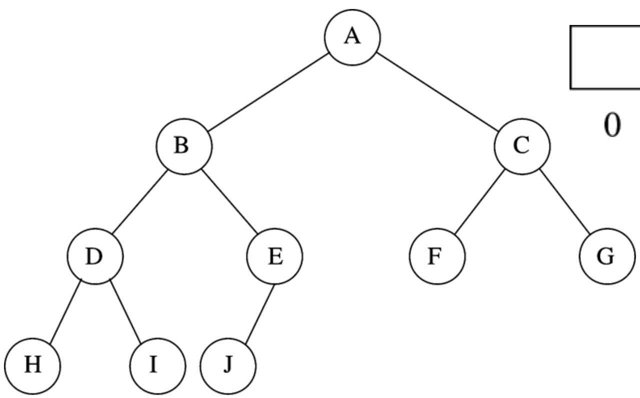


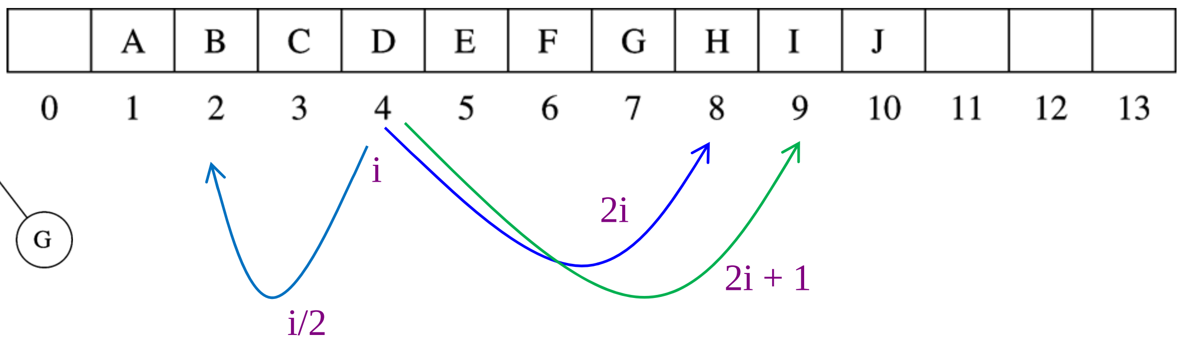
Not a Heap

Bhaskar Sardar, Information Technology Department, ***Jadavpur University, India***

Priority Queues and Heaps

- Given element at position i in the array
  - Left child(i) = at position 2i
  - Right child(i) = at position 2i + 1
  - Parent(i) = at position i/2





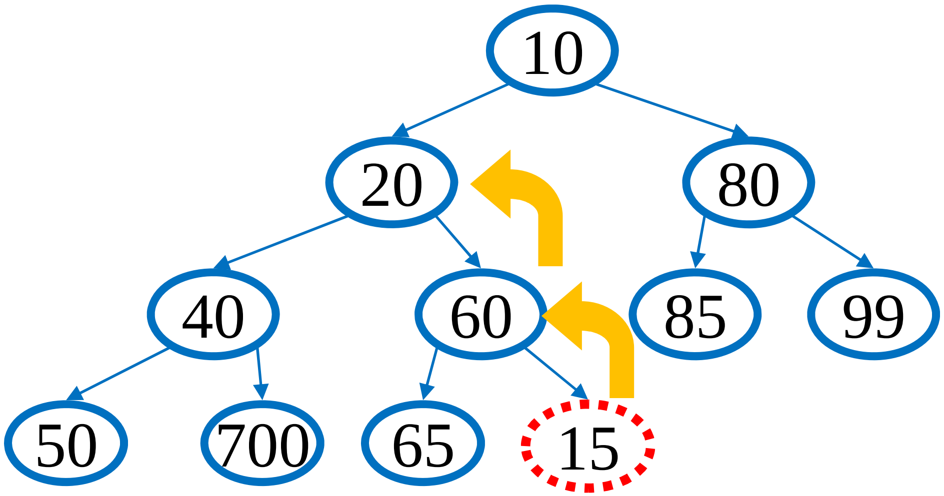
Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Priority Queues and Heaps

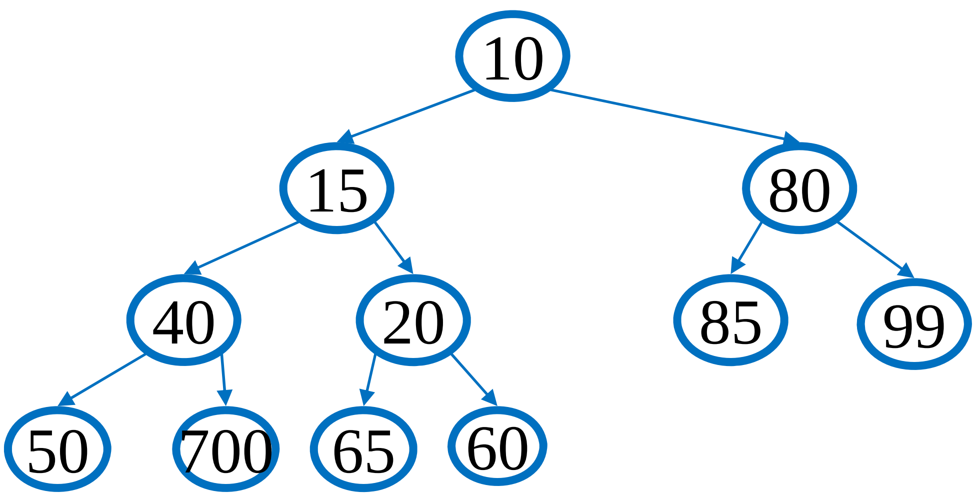
- **Insertion in a Heap**
  - Basic Idea:
    1. Put element at **“next”** leaf position
    2. **Restore heap property** by repeated exchange starting from inserted element until no longer needed

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Priority Queues and Heaps




```
void insert (int element)
{
      Pqueue [++n] = element;
      RestoreHeapUp (n);
}

void RestoreHeapUp (int pos)
{
      int element = Pqueue [pos];
      while ( Pqueue [pos/2] >= element)
      {
            Pqueue [pos ] = Pqueue [pos/2];
            pos = pos/2;
      }
      Pqueue [pos] = element;
}
```



Complexity = O (log n)

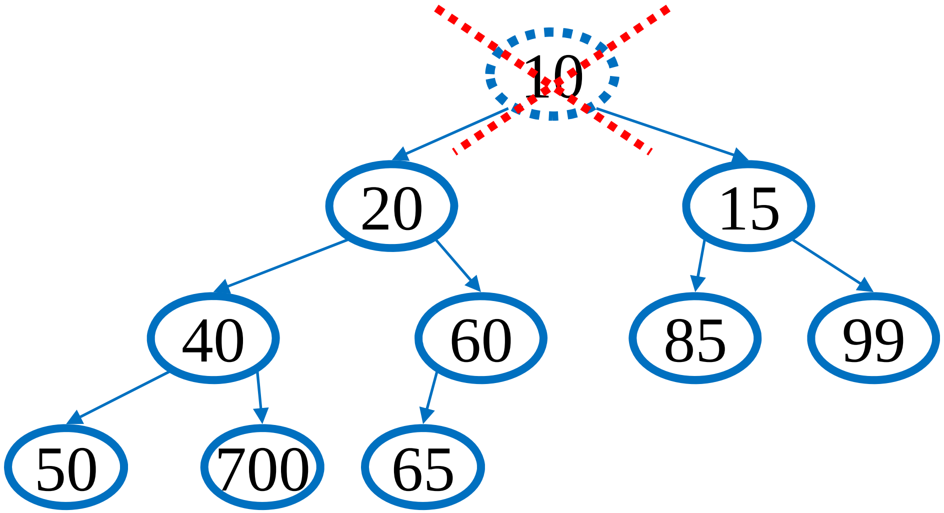
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

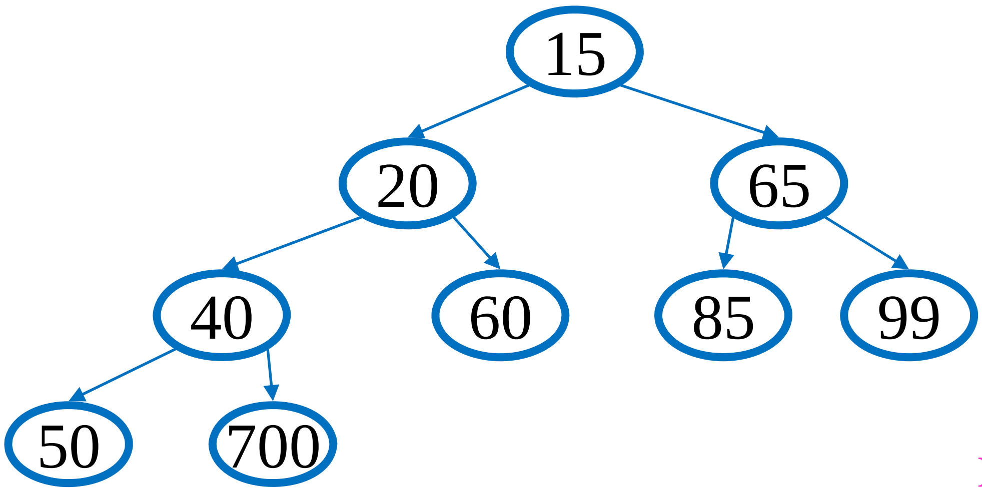
Priority Queues and Heaps

- **Deletion from a Heap**
  - Basic Idea:
    1. Remove **root** (that is always the min!)
    2. Put "last" leaf node at **root**
    3. **Restore heap property** by repeated exchange starting from deleted element until no longer needed

Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Priority Queues and Heaps






```
int Delete ()
{
        int element = Pqueue [ 1 ];
        Pqueue [ 1 ] = Pqueue [ n --];
        RestoreHeapDown ( 1 );
        return element;
}

void RestoreHeapDown ( int pos)
{
        int i, element = Pqueue [ pos ];
        while ( pos <= n/2)
        {
                i = 2*pos;
                if ( (i<n) && (Pqueue [i] > Pqueue [ i + 1] ) )
                        i ++;
                if ( element <= Pqueue [ i ] )
                        break;
                Pqueue [ pos ] = Pqueue [ i ];
                pos = i;
        }
        Pqueue [ pos ] = element;
}
```

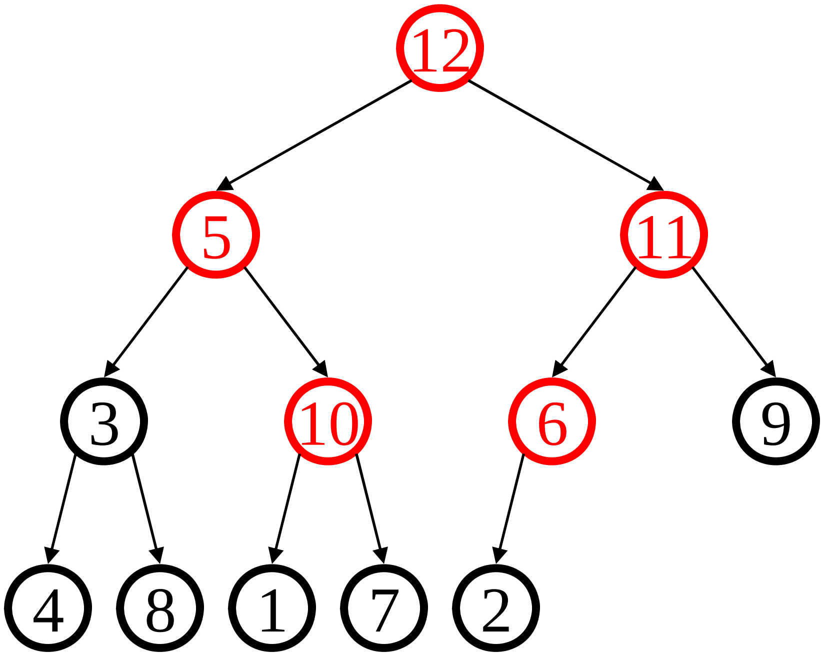
Complexity = O (log n)

Priority Queues and Heaps

- **Heap construction**

12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



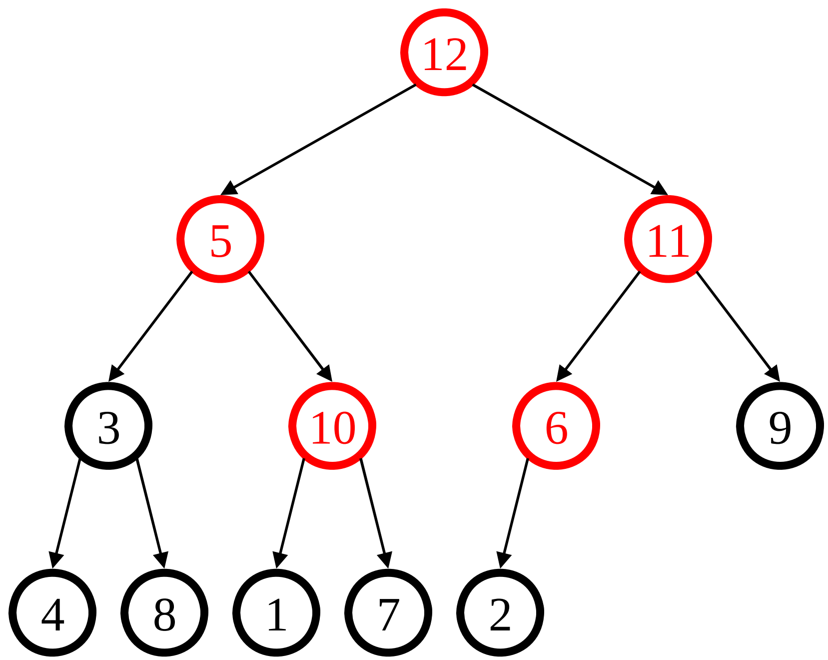
Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Priority Queues and Heaps

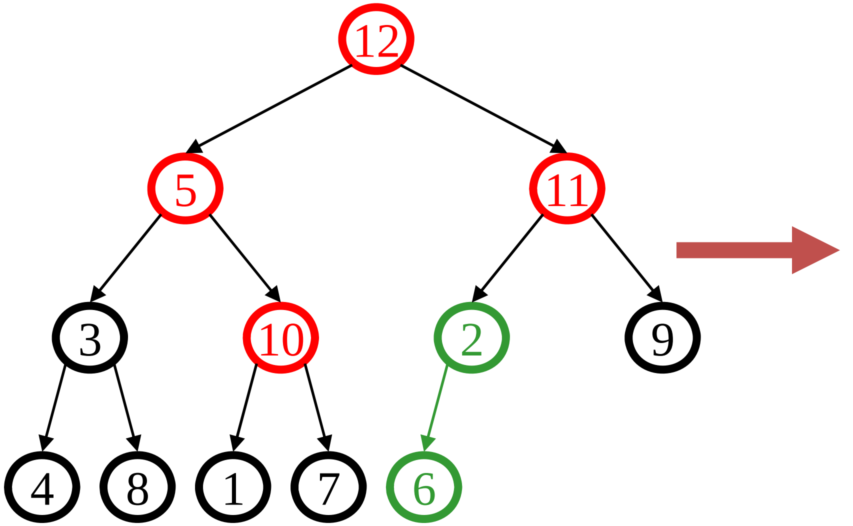
```
void BuildHeap ( )
{
        int i;
        for ( i = n/2; i > 0 ; i --)
        RestoreHeapDown ( i );
}
```

Complexity = O (n)

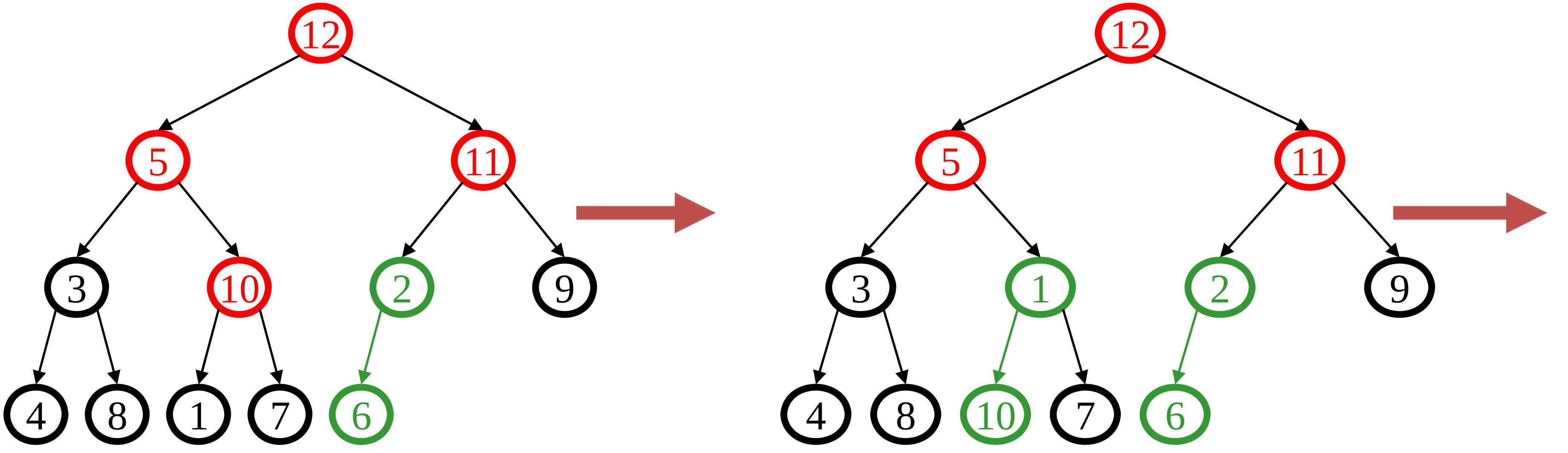
Priority Queues and Heaps



Priority Queues and Heaps

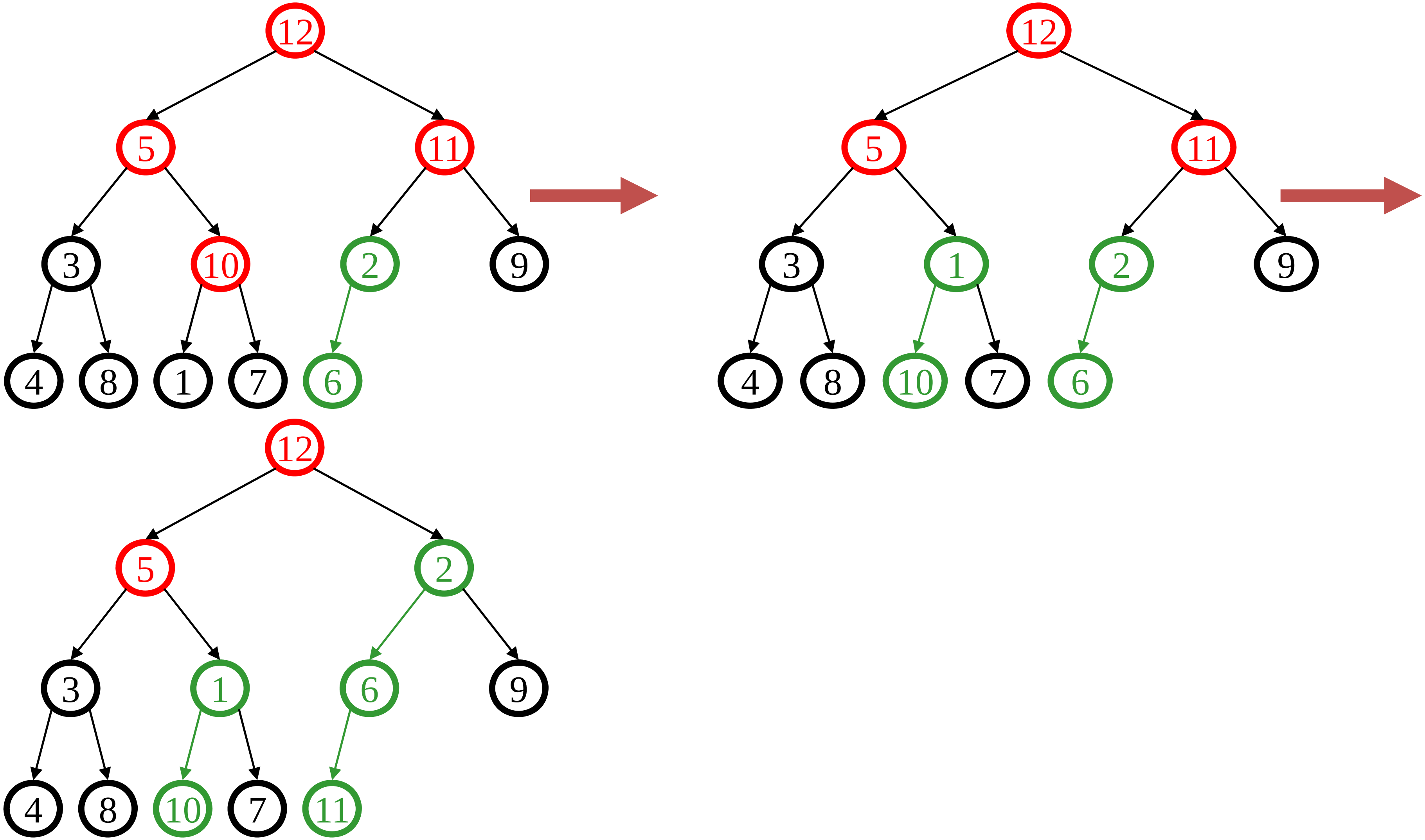


Priority Queues and Heaps



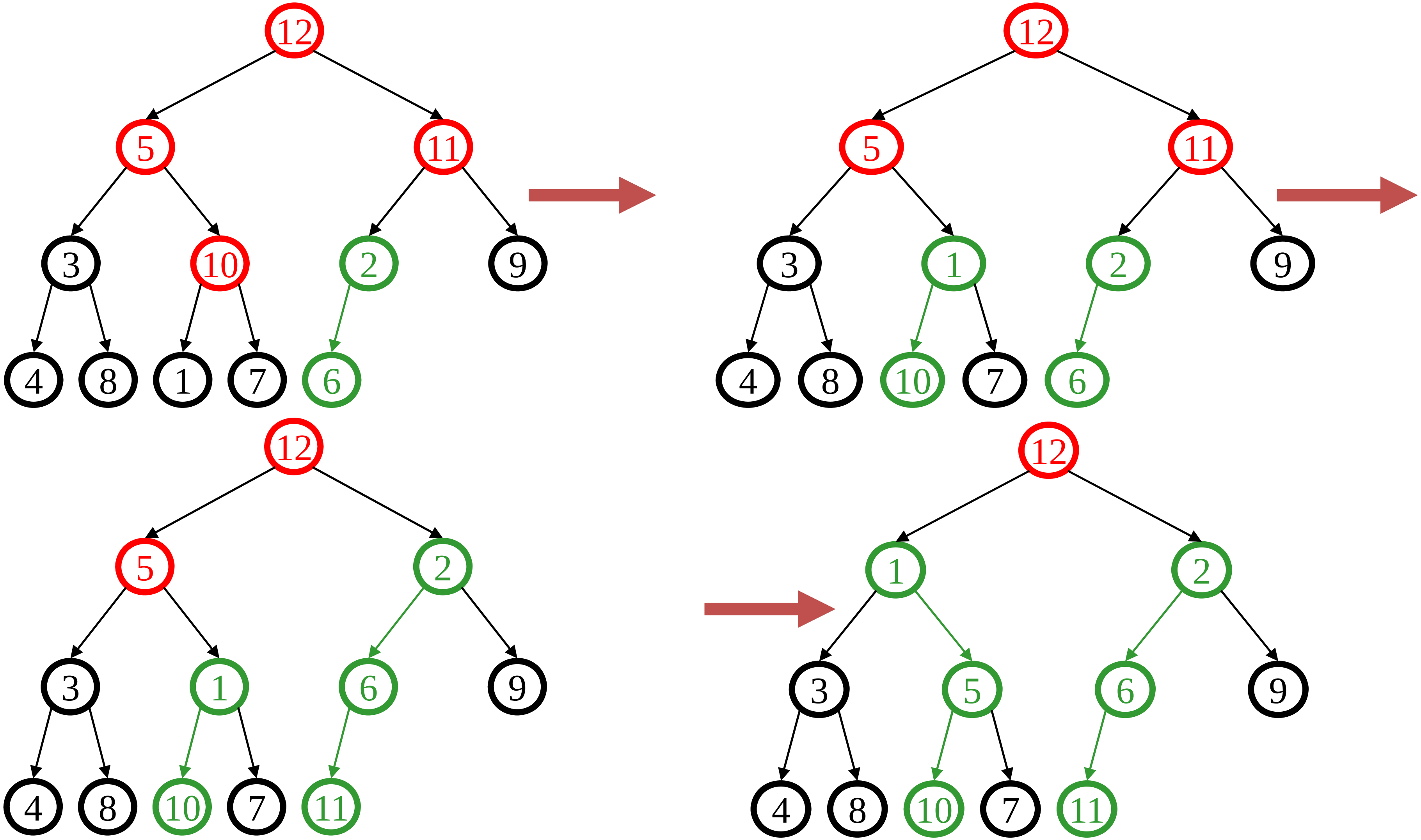
Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Priority Queues and Heaps



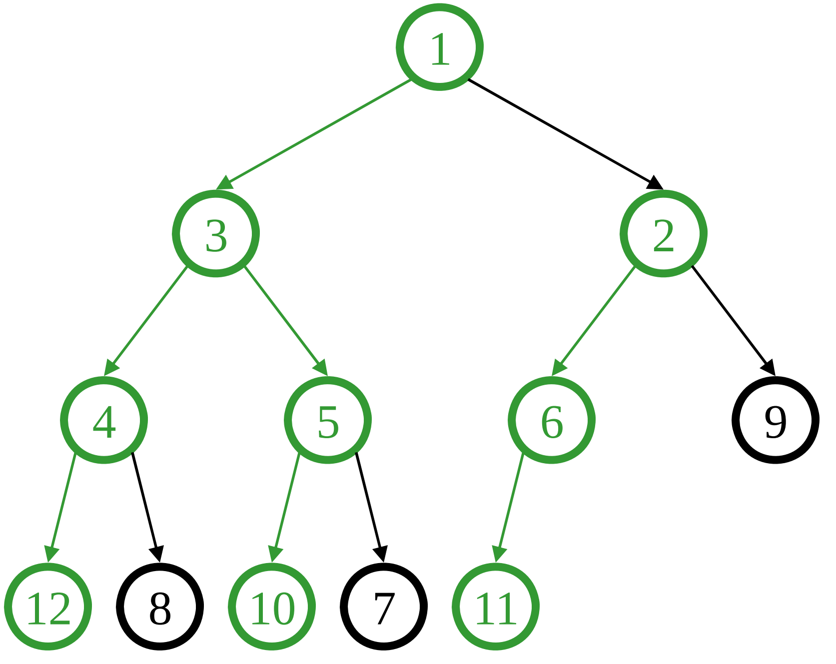
Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Priority Queues and Heaps



Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Priority Queues and Heaps



Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Heap Sort

- Given **BuildHeap()**, an in-place sorting algorithm is easily constructed:
  - Smallest element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement **Heap_size**
  - Restore heap property at A[1] by calling **RestoreHeapDown()**
  - Repeat, always swapping A[1] with A[heap_size]

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Heap Sort

```
Heapsort(A)
{
        BuildHeap(A);
        for (i = length(A) downto 2)
        {
                Swap(A[1], A[i]);
                heap_size(A) -= 1;
                RestoreHeapDown(A, 1);
        }
}
```


Bhaskar Sardar, Information Technology Department, Jadavpur University, India

Heap Sort

- The call to **BuildHeap()** takes O(n) time
- Each of the **n - 1** calls to **RestoreHeapDown()** takes **O(log n)** time
- Thus the total time taken by **HeapSort()**
  = **O(n) + (n - 1) O(log n)**
  = **O(n) + O(n log n)**
  = **O(n log n)**

Any Doubt ?

- Please feel free to write to me:

bhaskargit@yahoo.co.in