Unit-IV

Graphs

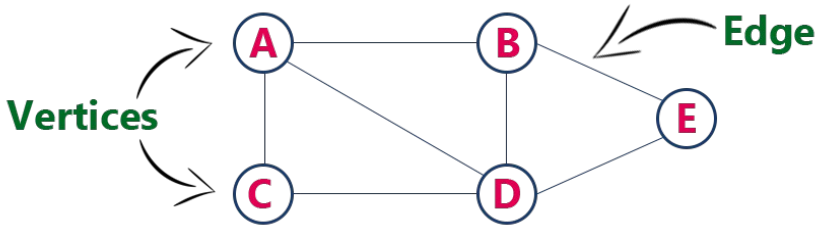
Introduction to Graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

- Graph is a collection of vertices and arcs in which vertices are connected with arcs
- Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Ex:
The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.
1. Undirected Edge - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. Directed Edge - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. Weighted Edge - A weighted egde is a edge with value (cost) on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

If a edge is directed, its first endpoint is said to be the origin of it.

Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

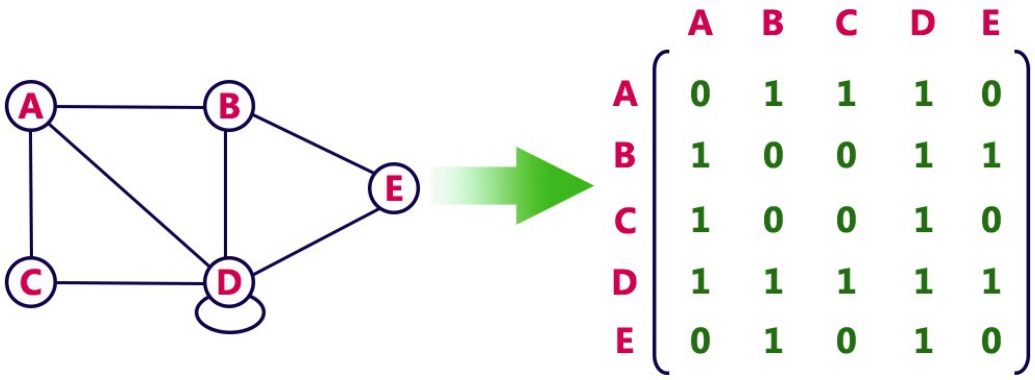
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

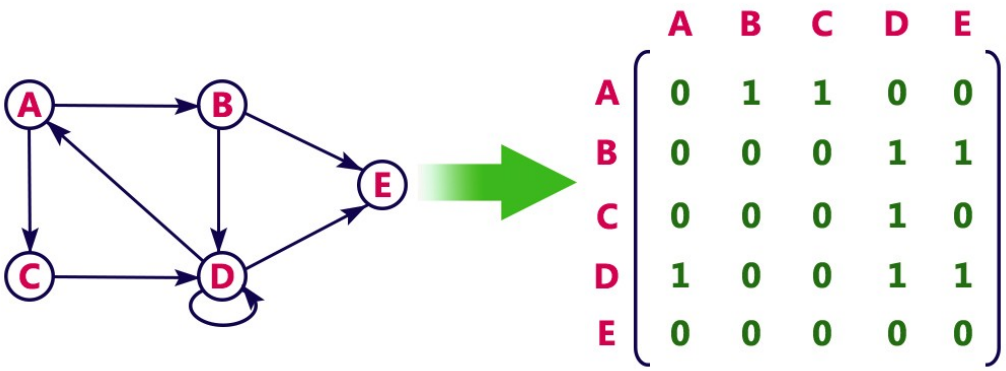
Adjacency Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



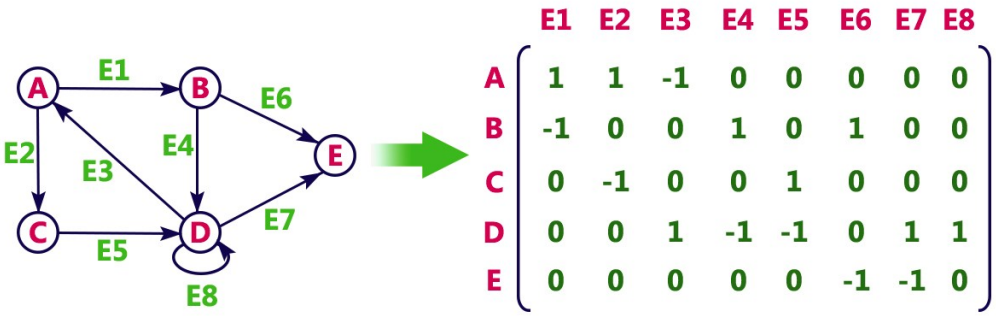
Directed graph representation...



Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

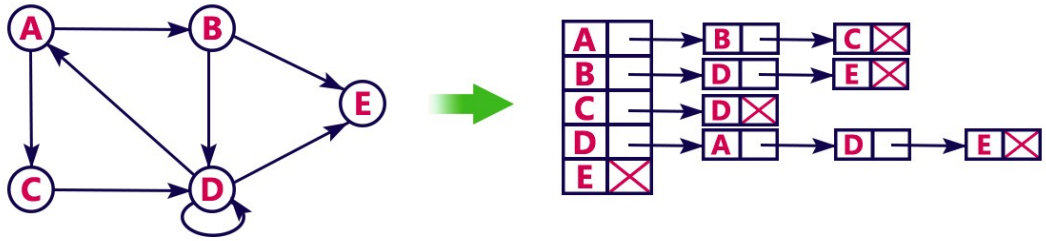
For example, consider the following directed graph representation...



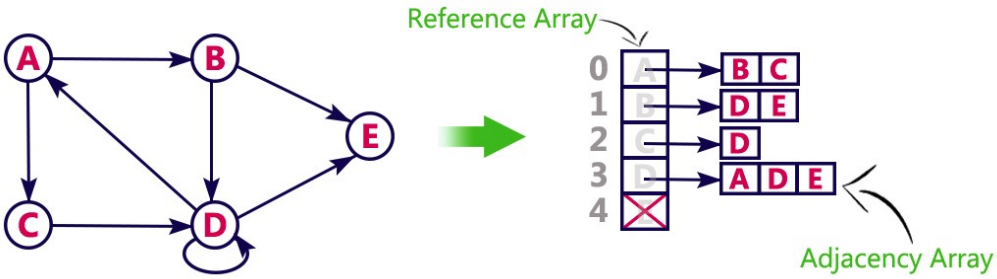
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

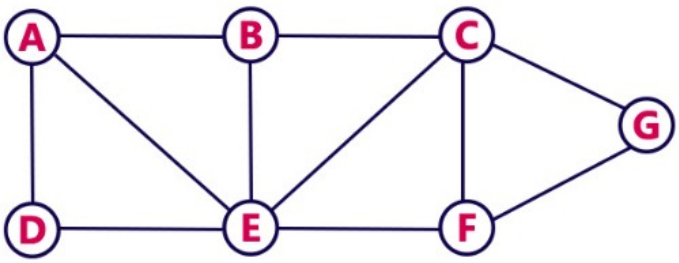
We use the following steps to implement DFS traversal...

1. Define a Stack of size total number of vertices in the graph.
2. Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
3. Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
5. When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
6. Repeat steps 3, 4 and 5 until stack becomes Empty.
7. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Note: **Back tracking** is coming back to the vertex from which we reached the current vertex.

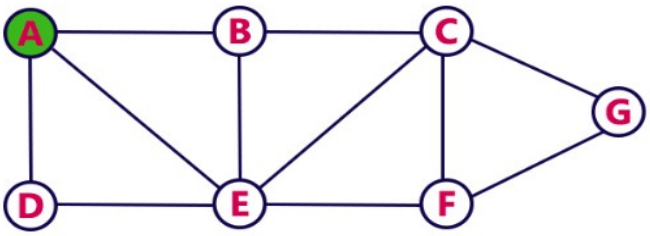
Ex:

Consider the following example graph to perform DFS traversal



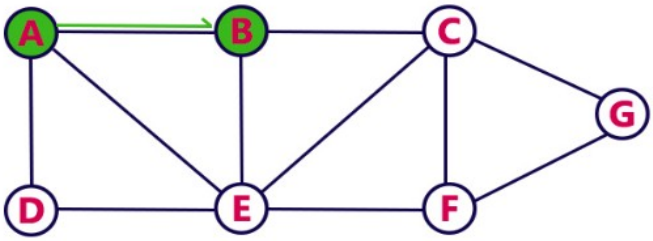
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



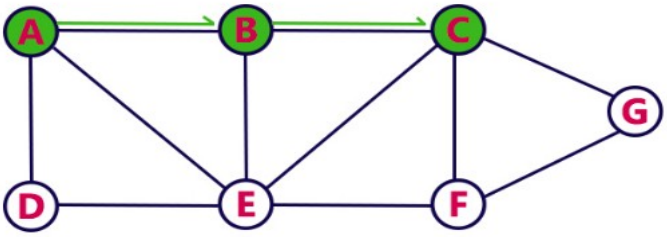
Step 2:

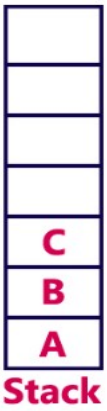
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

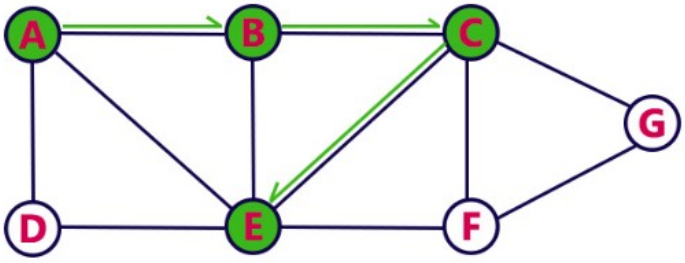
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





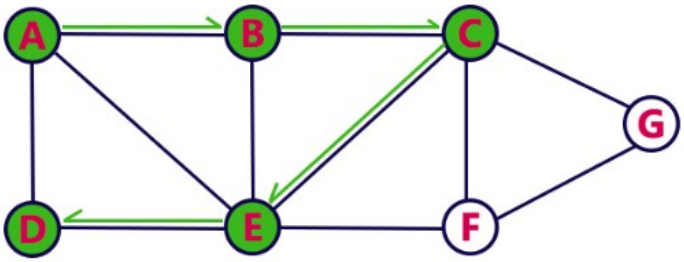
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



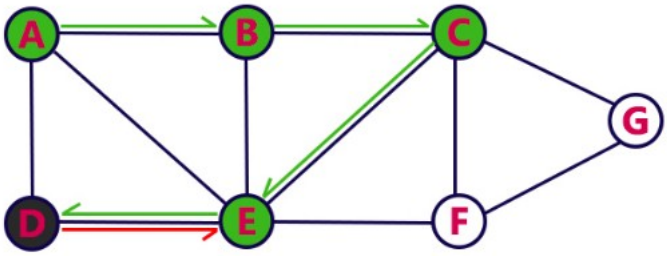
Step 5:

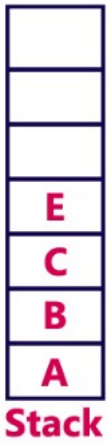
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



Step 6:

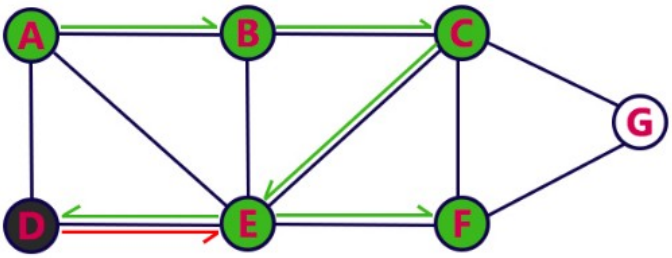
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.





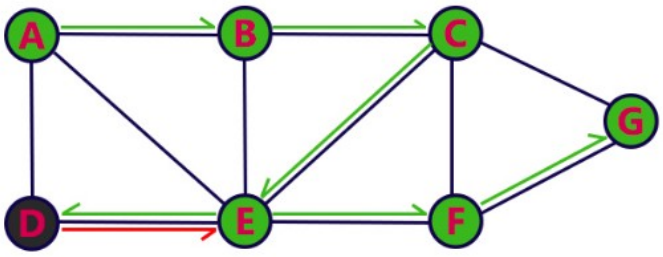
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



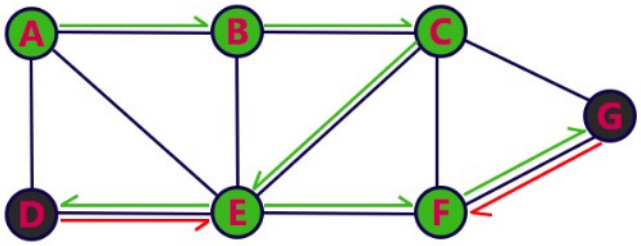
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



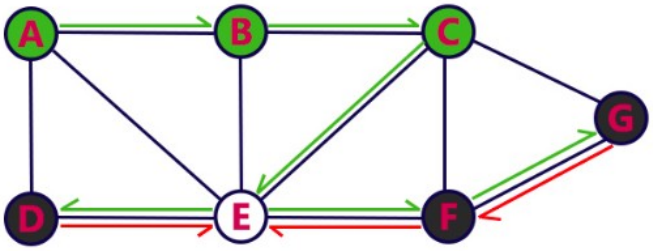
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



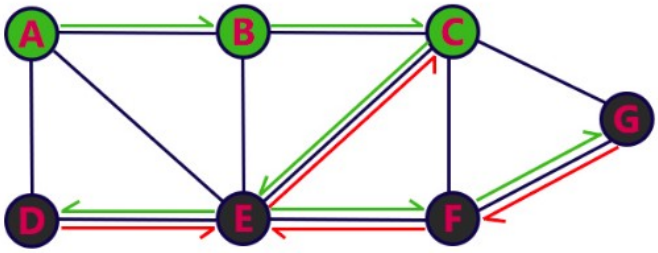
Step 10:

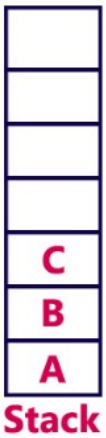
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

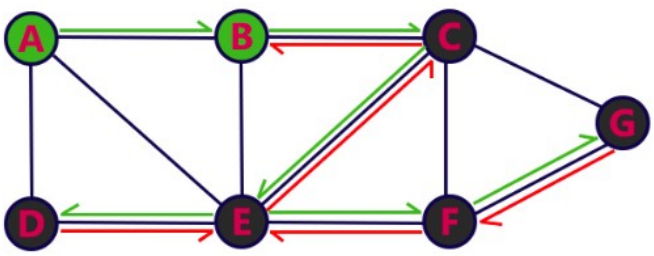
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





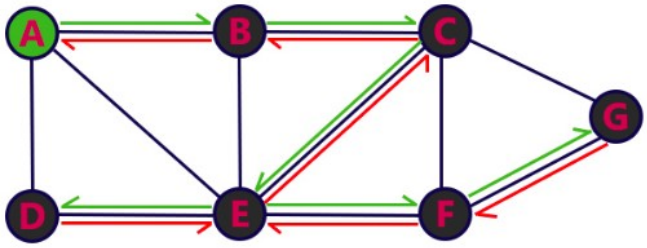
Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



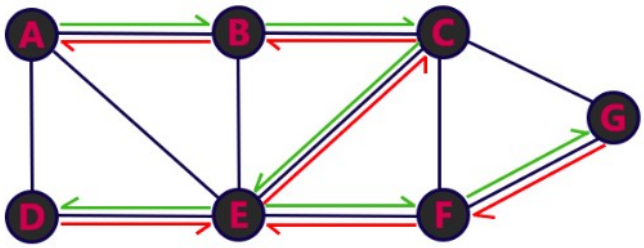
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

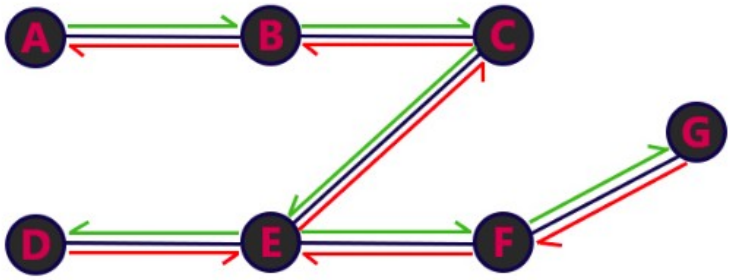


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

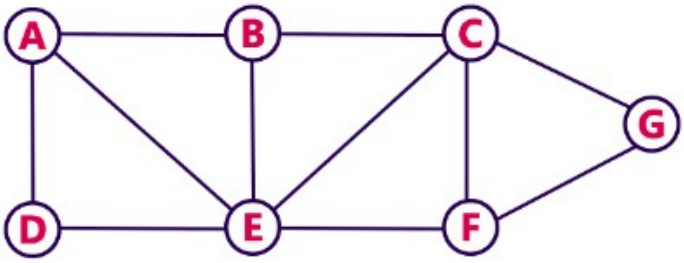
We use the following steps to implement BFS traversal...

1. Define a Queue of size total number of vertices in the graph.
2. Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

3. Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
5. Repeat steps 3 and 4 until queue becomes empty.
6. When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

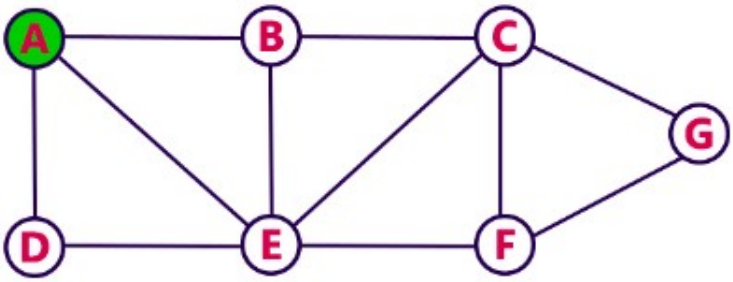
Ex:

Consider the following example graph to perform BFS traversal



Step 1:

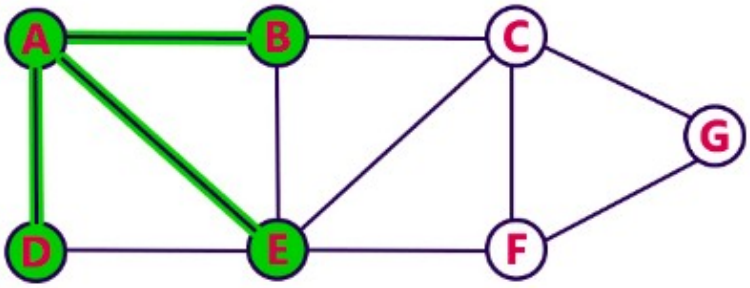
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

Step 2:

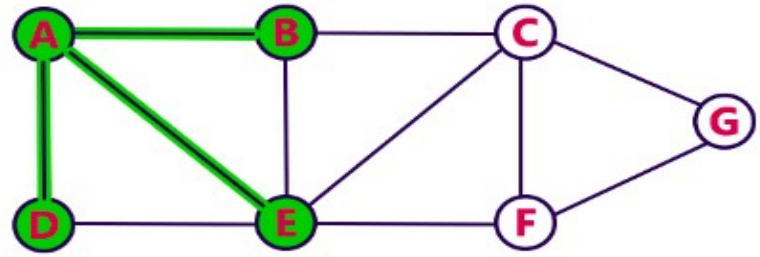
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

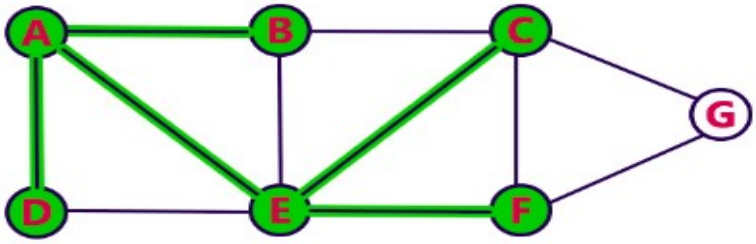


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

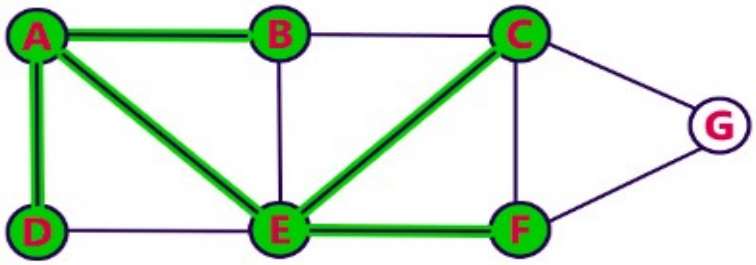


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

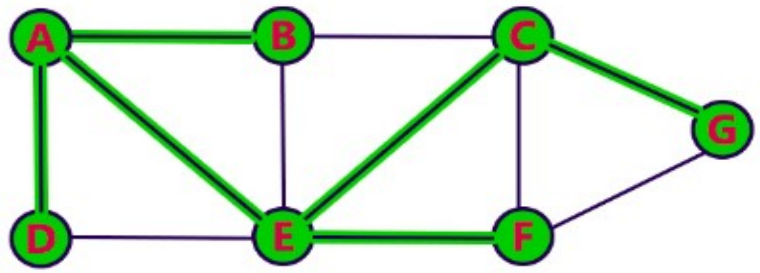


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

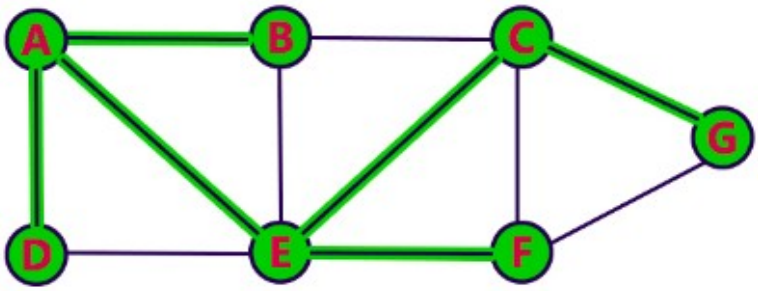


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

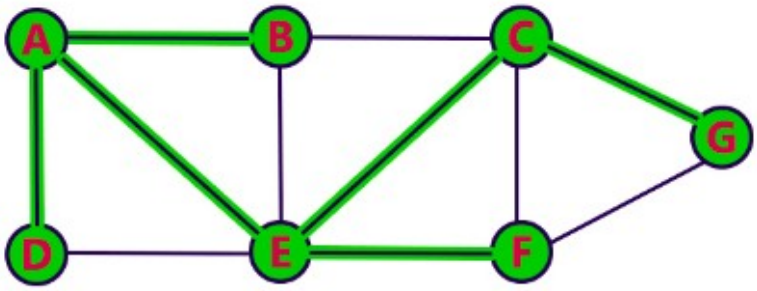


Queue

						G

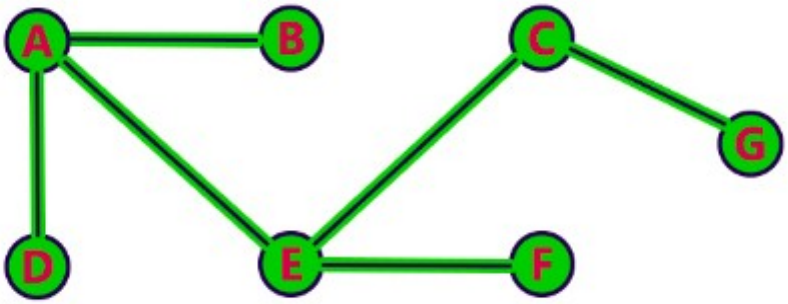
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Sorting

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion Sort

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

The insertion sort algorithm is performed using the following steps...

1. Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
2. Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
3. Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Ex:

Consider the following unsorted list of elements...

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Move the first element 15 from unsorted portion to sorted portion of the list.

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.


```
//Insertion Sorting
#include<stdio.h>
void main()
{
  int size, i, j, temp, list[100];
  printf("Enter the size of the list: ");
  scanf("%d", &size);
  printf("Enter %d integer values: ", size);
  for (i = 0; i < size; i++)
    scanf("%d", &list[i]);

  //Insertion sort logic
  for (i = 1; i < size; i++) {
    temp = list[i];
    j = i - 1;
    while ((temp < list[j]) && (j >= 0)) {
      list[j + 1] = list[j];
      j = j - 1;
    }
    list[j + 1] = temp;
  }
  printf("List after Sorting is: ");
  for (i = 0; i < size; i++)
    printf(" %d", list[i]);
}
```

Selection Sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

The selection sort algorithm is performed using the following steps...

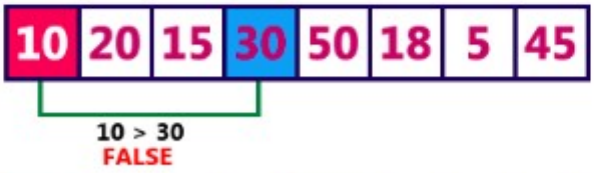
1. Select the first element of the list (i.e., Element at first position in the list).
2. Compare the selected element with all the other elements in the list.
3. In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
4. Repeat the same procedure with element in the next position in the list till the entire list is sorted.

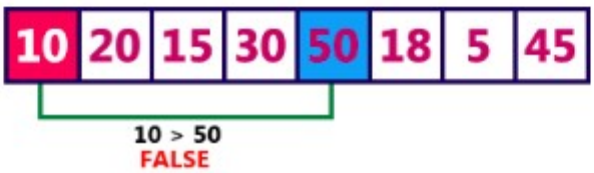
Ex:

Consider the following unsorted list of elements...

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.





List after 1st iteration

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

Iteration #6

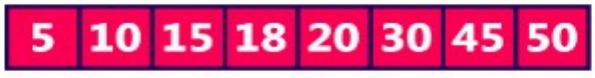
Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration



Final sorted list

```
//Selection Sort
#include<stdio.h>

void main()
{
  int size,i,j,temp,list[100];
  printf("Enter the size of the List: ");
  scanf("%d",&size);
  printf("Enter %d integer values: ",size);
  for(i=0; i<size; i++)
     scanf("%d",&list[i]);

  //Selection sort logic

  for(i=0; i<size; i++){
     for(j=i+1; j<size; j++){
          if(list[i] > list[j])
              {
              temp=list[i];
              list[i]=list[j];
              list[j]=temp;
          }
     }
  }

  printf("List after sorting is: ");
  for(i=0; i<size; i++)
     printf(" %d",list[i]);
}
```

Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

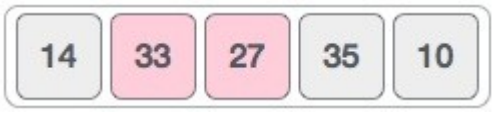
Ex:

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

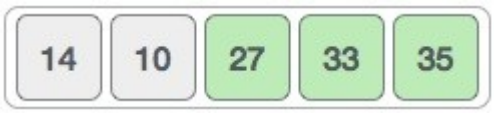
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

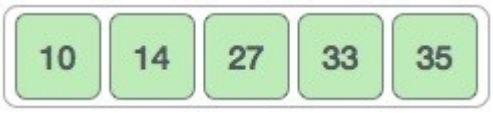
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.




```
//Bubble Sort
#include<stdio.h>

int main()
{
	int a[50],n,i,j,temp;
	printf("Enter the size of array: ");
	scanf("%d",&n);
	printf("Enter the array elements: ");

	for(i=0;i<n;++i)
	scanf("%d",&a[i]);

	for(i=1;i<n;++i)
	for(j=0;j<(n-i);++j)
		if(a[j]>a[j+1])
		{
			temp=a[j];
			a[j]=a[j+1];
			a[j+1]=temp;
		}

	printf("\nArray after sorting: ");
	for(i=0;i<n;++i)
	printf("%d ",a[i]);

	return 0;
}
```

Heap Sort

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

Max Heap

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure some times also called as Binary Heap.

There are two types of heap data structures and they are as follows...

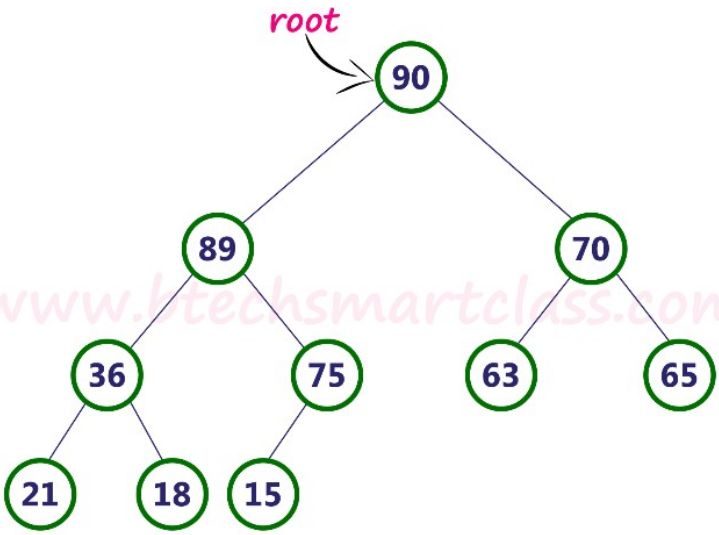
1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...
1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.
2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

Ex:



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations

Finding Maximum Value

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

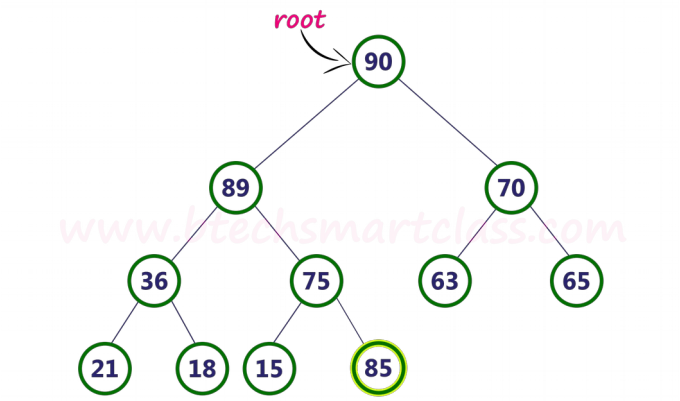
Insertion

Insertion Operation in max heap is performed as follows...

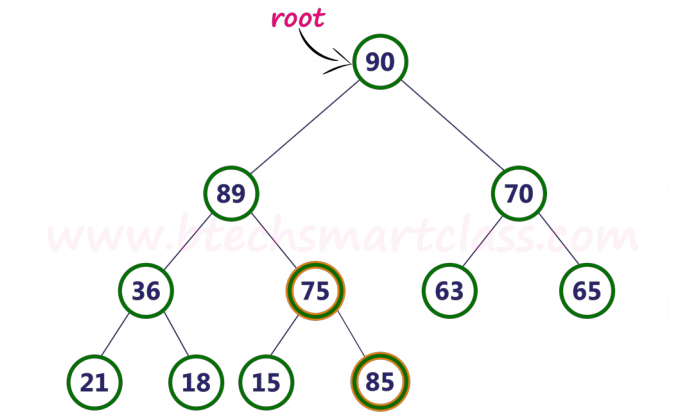
1. Insert the **newNode** as **last leaf** from left to right.
2. Compare **newNode value** with its **Parent node**.
3. If **newNode value is greater** than its parent, then **swap** both of them.
4. Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

Ex:
Consider the above max heap. **Insert a new node with value 85.**

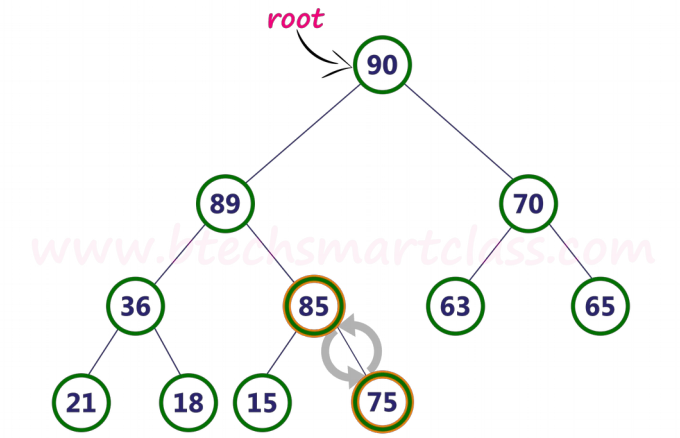
Step 1 - Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



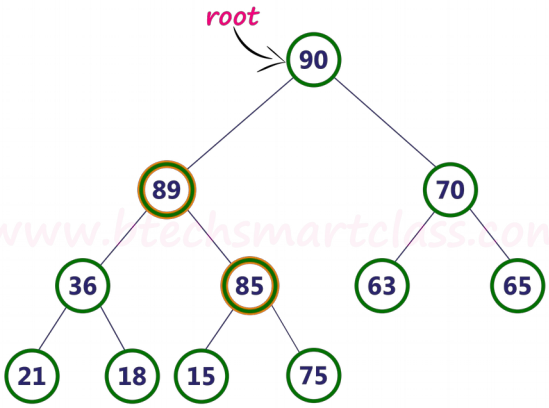
Step 2 - Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



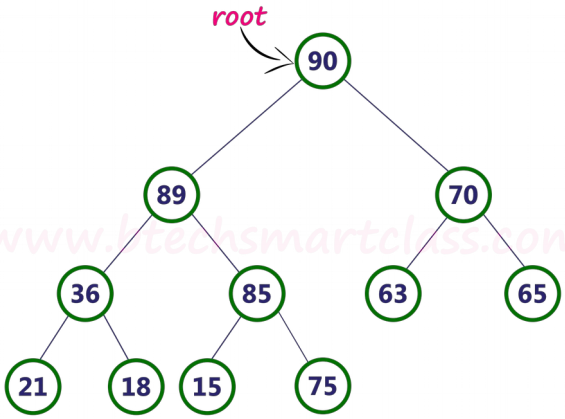
Step 3 - Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...



Step 4 - Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion

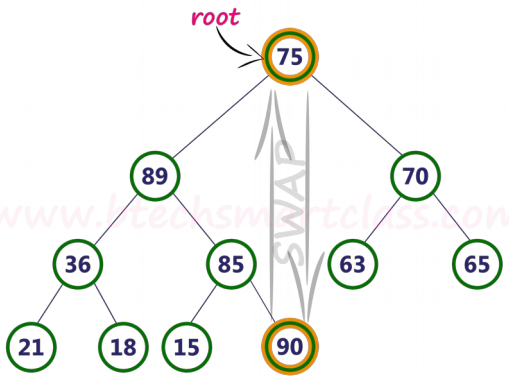
In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

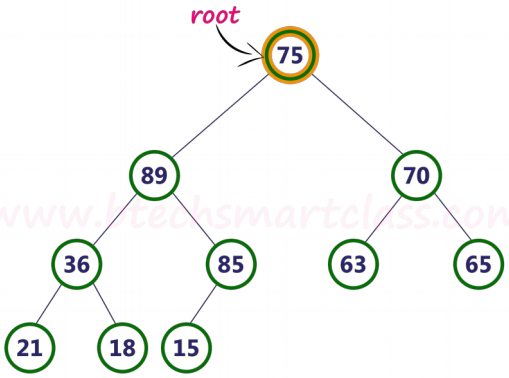
1. **Swap** the **root** node with **last** node in max heap
2. **Delete** last node.
3. Now, compare **root value** with its **left child value**.
4. If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto **Step 6**
5. If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
6. If **root value is larger** than its left child, then compare **root value** with its **right child** value.
7. If **root value is smaller** than its **right child**, then **swap root** with **right child** otherwise **stop the process**.
8. Repeat the same until root node fixes at its exact position.

Ex:
Consider the above max heap. **Delete root node (90) from the max heap.**

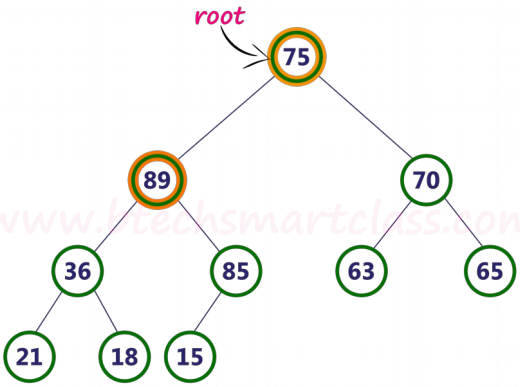
Step 1 - **Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



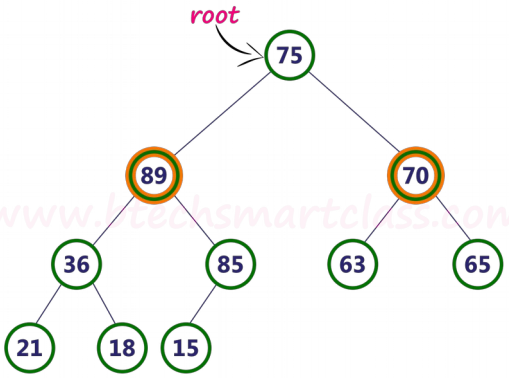
Step 2 - **Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



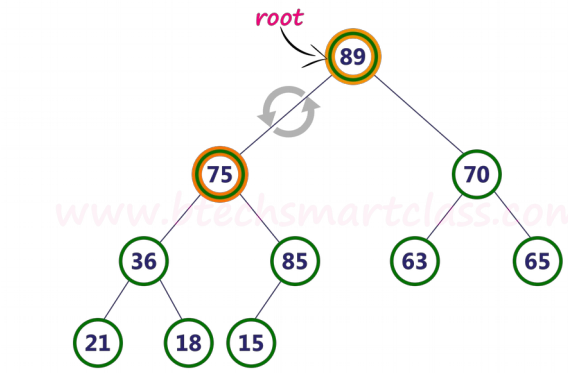
Step 3 - Compare **root node (75)** with its **left child (89)**.



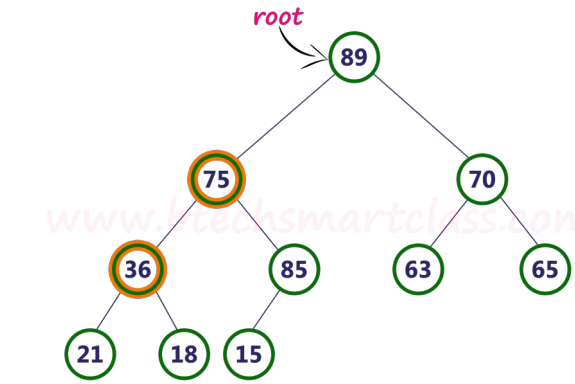
Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



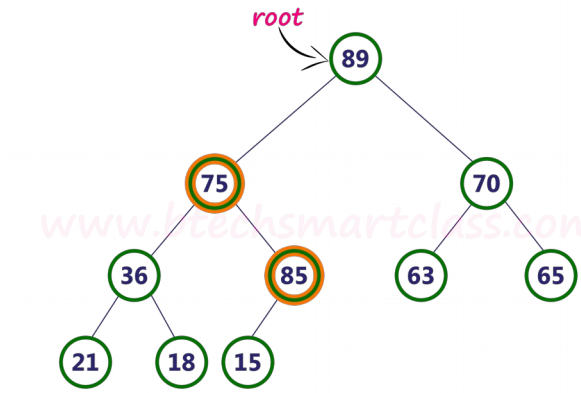
Step 4 - Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



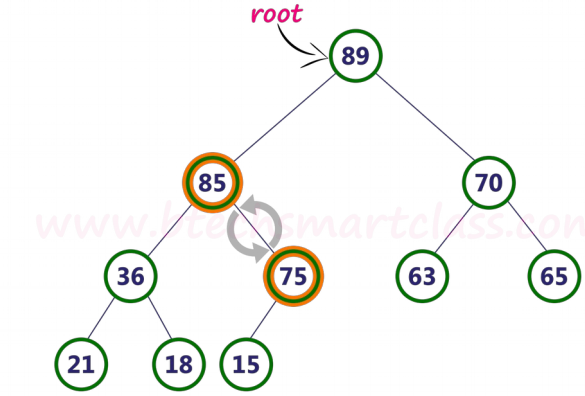
Step 5 - Now, again compare **75** with its **left child (36)**.



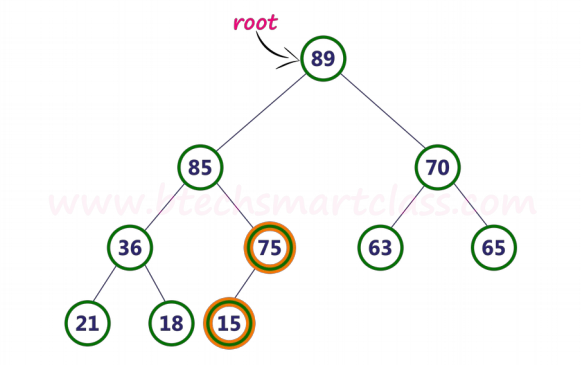
Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



Step 6 - Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

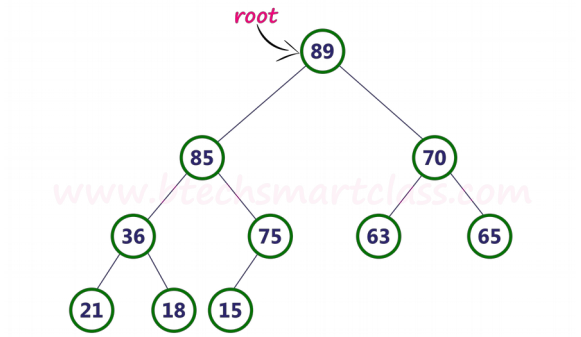


Step 7 - Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

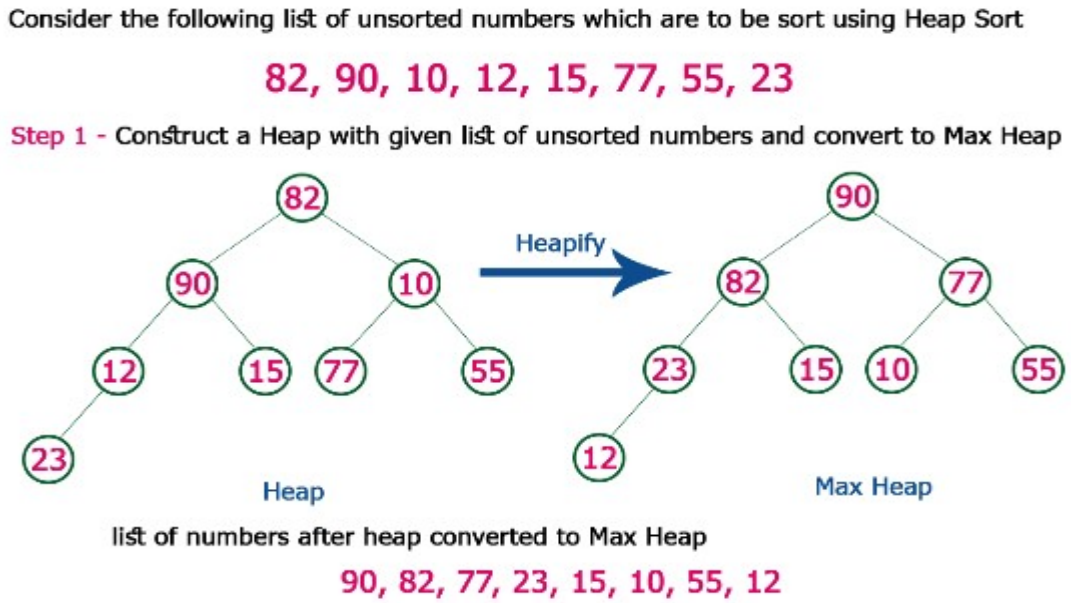
Finally, max heap after deleting root node (**90**) is as follows...



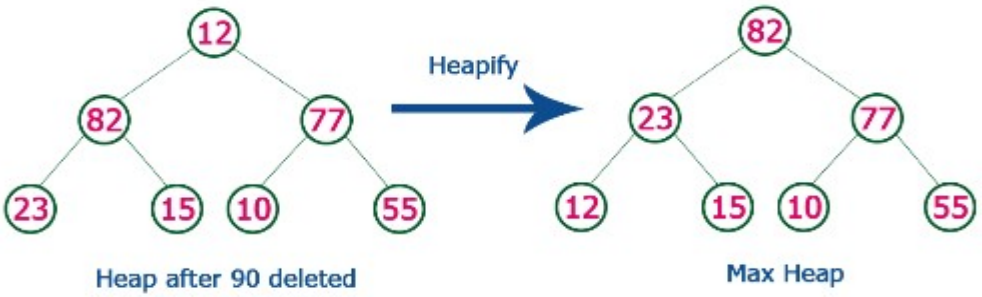
The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

1. Construct a **Binary Tree** with given list of Elements.
2. Transform the Binary Tree into **Min Heap.**
3. Delete the root element from Min Heap using **Heapify** method.
4. Put the deleted element into the Sorted list.
5. Repeat the same until Min Heap becomes empty.
6. Display the sorted list.

Ex:



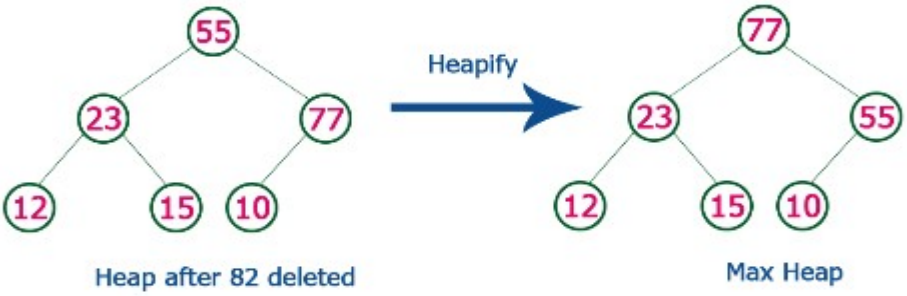
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

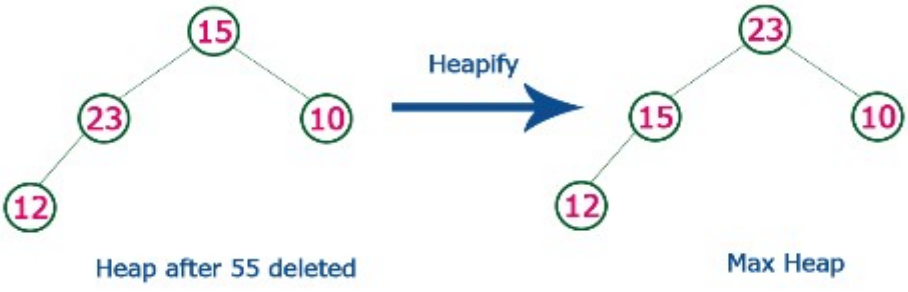
12, 55, 77, 23, 15, 10, **82**, **90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

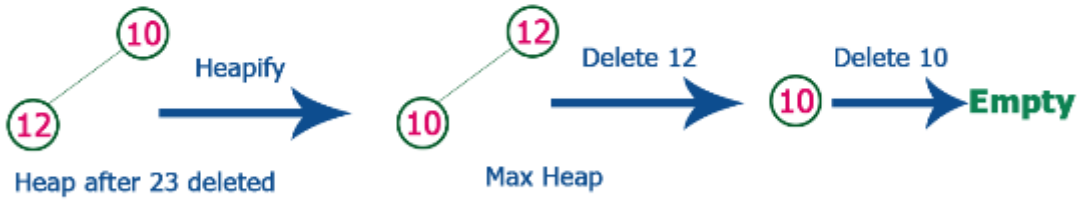
12, 15, 10, 23, **55**, **77**, **82**, **90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Merge Sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Steps:

MergeSort(arr[], l, r)

If r > l

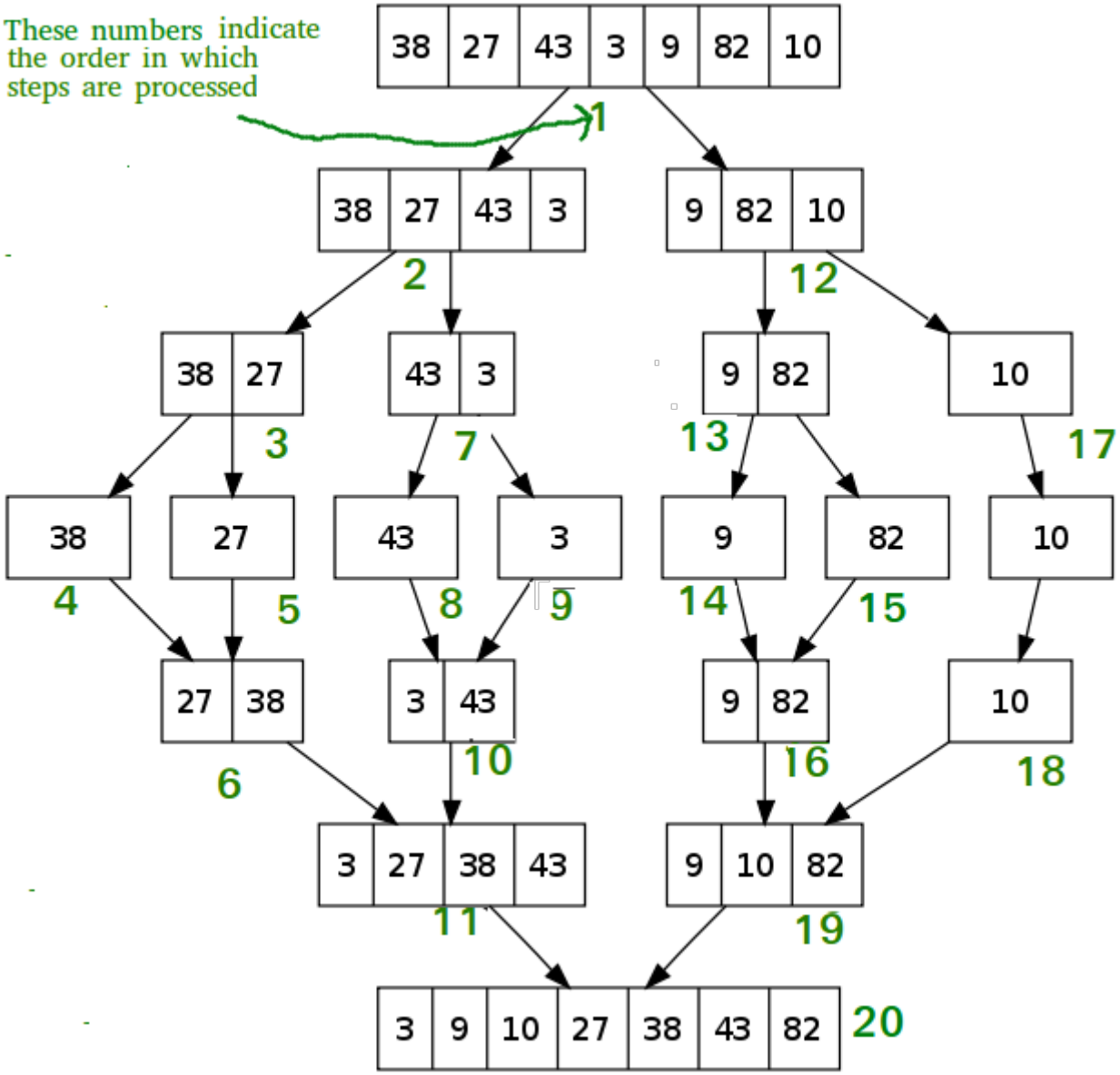
1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2

2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

Ex:


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

```
///Merge Sort
#include<stdio.h>
 void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
 int main()
{
        int a[30],n,i;
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter array elements:");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        mergesort(a,0,n-1);
        printf("\nSorted array is :");
        for(i=0;i<n;i++)
        printf("%d ",a[i]);
        return 0;
}
void mergesort(int a[],int i,int j)
{
        int mid;
        if(i<j)
        {
        mid=(i+j)/2;
        mergesort(a,i,mid);                     //left recursion
        mergesort(a,mid+1,j);       //right recursion
        merge(a,i,mid,mid+1,j);     //merging of two sorted sub-arrays
        }
}
void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[50];       //array used for merging
        int i,j,k;
        i=i1;       //beginning of the first list
        j=i2;       //beginning of the second list
        k=0;
        while(i<=j1 && j<=j2)      //while elements in both lists
        {
        if(a[i]<a[j])
                temp[k++]=a[i++];
        else
                temp[k++]=a[j++];
        }

        while(i<=j1)        //copy remaining elements of the first list
        temp[k++]=a[i++];
        while(j<=j2)        //copy remaining elements of the second list
        temp[k++]=a[j++];
        //Transfer elements from temp[] back to a[]
        for(i=i1,j=0;i<=j2;i++,j++)
        a[i]=temp[j];
}
```

External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Prerequisite:
MergeSort : Used for sort individual runs (a run is part of file that is small enough to fit in main memory)
Merge K Sorted Arrays : Used to merge sorted runs.

We now consider the problem of sorting collections of records too large to fit in main memory. Because the records must reside in peripheral or external memory, such sorting methods are called ***eternal sorts***. This is in contrast to ***internal sorts***, which assume that the records to be sorted are stored in main memory. Sorting large collections of records is central to many applications, such as processing payrolls and other large business databases. As a consequence, many external sorting algorithms have been devised. Years ago, sorting algorithm designers sought to optimize the use of specific hardware configurations, such as multiple tape or *disk drives*. Most computing today is done on personal computers and low-end workstations with relatively powerful CPUs, but only one or at most two disk drives. The techniques presented here are geared toward optimized processing on a single disk drive. This approach allows us to cover the most important issues in external sorting while skipping many less important machine-dependent details.

When a collection of records is too large to fit in ***main memory***, the only practical way to sort it is to read some records from disk, do some rearranging, then write them back to disk. This process is repeated until the file is sorted, with each record read perhaps many times. Given the high cost of ***disk I/O***, it should come as no surprise that the primary goal of an external sorting algorithm is to minimize the number of times information must be read from or written to disk. A certain amount of additional CPU processing can profitably be traded for reduced disk access.

Before discussing external sorting techniques, consider again the basic model for accessing information from disk. The file to be sorted is viewed by the programmer as a sequential series of fixed-size ***blocks***. Assume (for simplicity) that each block contains the same number of fixed-size data records. Depending on the application, a record might be only a few bytes—composed of little or nothing more than the key—or might be hundreds of bytes with a relatively small key field. Records are assumed not to cross block boundaries. These assumptions can be relaxed for special-purpose sorting applications, but ignoring such complications makes the principles clearer.

Recall that a ***sector*** is the basic unit of I/O. In other words, all disk reads and writes are for one or more complete sectors. Sector sizes are typically a power of two, in the range 512 to 16K bytes, depending on the

operating system and the size and speed of the disk drive. The block size used for external sorting algorithms should be equal to or a multiple of the sector size.

Under this model, a sorting algorithm reads a block of data into a buffer in main memory, performs some processing on it, and at some future time writes it back to disk. *Recall that* reading or writing a block from disk takes on the order of one million times longer than a memory access. Based on this fact, we can reasonably expect that the records contained in a single block can be sorted by an internal sorting algorithm such as ***Quicksort*** in less time than is required to read or write the block.

Under good conditions, reading from a file in sequential order is more efficient than reading blocks in random order. Given the significant impact of seek time on disk access, it might seem obvious that sequential processing is faster. However, it is important to understand precisely under what circumstances sequential file processing is actually faster than random access, because it affects our approach to designing an external sorting algorithm.

Efficient sequential access relies on seek time being kept to a minimum. The first requirement is that the blocks making up a file are in fact stored on disk in sequential order and close together, preferably filling a small number of contiguous tracks. At the very least, the number of extents making up the file should be small. Users typically do not have much control over the layout of their file on disk, but writing a file all at once in sequential order to a disk drive with a high percentage of free space increases the likelihood of such an arrangement.

The second requirement is that the disk drive's I/O head remain positioned over the file throughout sequential processing. This will not happen if there is competition of any kind for the I/O head. For example, on a multi-user time-shared computer the sorting process might compete for the I/O head with the processes of other users. Even when the sorting process has sole control of the I/O head, it is still likely that sequential processing will not be efficient. Imagine the situation where all processing is done on a single disk drive, with the typical arrangement of a single bank of read/write heads that move together over a stack of platters. If the sorting process involves reading from an input file, alternated with writing to an output file, then the I/O head will continuously seek between the input file and the output file. Similarly, if two input files are being processed simultaneously (such as during a merge process), then the I/O head will continuously seek between these two files.

The moral is that, with a single disk drive, there often is no such thing as efficient sequential processing of a data file. Thus, a sorting algorithm might be more efficient if it performs a smaller number of non-sequential disk operations rather than a larger number of logically sequential disk operations that require a large number of seeks in practice.

As mentioned previously, the record size might be quite large compared to the size of the key. For example, payroll entries for a large business might each store hundreds of bytes of information including the name, ID, address, and job title for each employee. The sort key might be the ID number, requiring only a few bytes. The simplest sorting algorithm might be to process such records as a whole, reading the entire record whenever it is processed. However, this will greatly increase the amount of I/O required, because only a relatively few records will fit into a single disk block. Another alternative is to do a ***key sort***. Under this method, the keys are all read and stored together in an ***index file***, where each key is stored along with a pointer indicating the position of the corresponding record in the original data file. The key and pointer combination should be substantially smaller

than the size of the original record; thus, the index file will be much smaller than the complete data file. The index file will then be sorted, requiring much less I/O because the index records are smaller than the complete records.

Once the index file is sorted, it is possible to reorder the records in the original database file. This is typically not done for two reasons. First, reading the records in sorted order from the record file requires a random access for each record. This can take a substantial amount of time and is only of value if the complete collection of records needs to be viewed or processed in sorted order (as opposed to a search for selected records). Second, database systems typically allow searches to be done on multiple keys. For example, today's processing might be done in order of ID numbers. Tomorrow, the boss might want information sorted by salary. Thus, there might be no single "sorted" order for the full record. Instead, multiple index files are often maintained, one for each sort key.

Simple Approaches to External Sorting

If your operating system supports virtual memory, the simplest "external" sort is to read the entire file into virtual memory and run an internal sorting method such as Quicksort. This approach allows the virtual memory manager to use its normal buffer pool mechanism to control disk accesses. Unfortunately, this might not always be a viable option. One potential drawback is that the size of virtual memory is usually limited to something much smaller than the disk space available. Thus, your input file might not fit into virtual memory. Limited virtual memory can be overcome by adapting an internal sorting method to make use of your own buffer pool.

A more general problem with adapting an internal sorting algorithm to external sorting is that it is not likely to be as efficient as designing a new algorithm with the specific goal of minimizing disk I/O. Consider the simple adaptation of Quicksort to use a buffer pool. Quicksort begins by processing the entire array of records, with the first partition step moving indices inward from the two ends. This can be implemented efficiently using a buffer pool. However, the next step is to process each of the subarrays, followed by processing of sub-subarrays, and so on. As the subarrays get smaller, processing quickly approaches random access to the disk drive. Even with maximum use of the buffer pool, Quicksort still must read and write each record $\log n$ times on average. We can do much better. Finally, even if the virtual memory manager can give good performance using a standard Quicksort, this will come at the cost of using a lot of the system's working memory, which will mean that the system cannot use this space for other work. Better methods can save time while also using less memory.

Our approach to external sorting is derived from the Mergesort algorithm. The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass. The first pass merges sublists of size 1 into sublists of size 2; the second pass merges the sublists of size 2 into sublists of size 4; and so on. A sorted sublist is called a ***run***. Thus, each pass is merging pairs of runs to form longer runs. Each pass copies the contents of the file to another file.

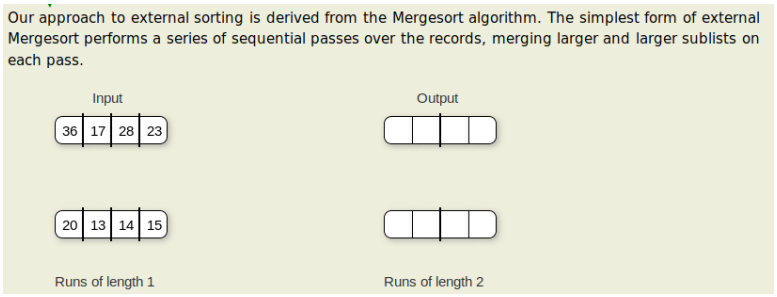
Steps:

1. Split the original file into two equal-sized ***run files***.
2. Read one block from each run file into input buffers.
3. Take the first record from each input buffer, and write a run of length two to an output buffer in sorted order.
4. Take the next record from each input buffer, and write a run of length two to a second output buffer in sorted order.

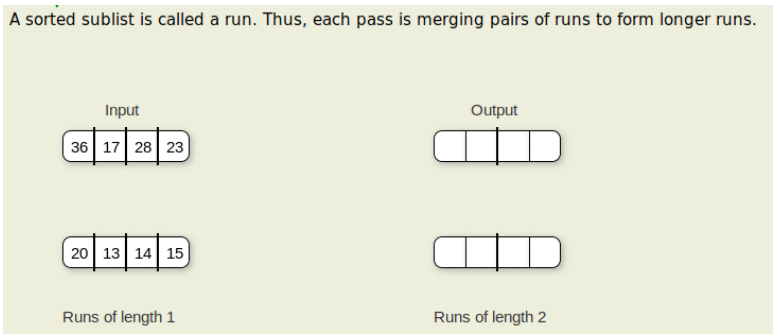
5. Repeat until finished, alternating output between the two output run buffers. Whenever the end of an input block is reached, read the next block from the appropriate input file. When an output buffer is full, write it to the appropriate output file.
6. Repeat steps 2 through 5, using the original output files as input files. On the second pass, the first two records of each input run file are already in sorted order. Thus, these two runs may be merged and output as a single run of four elements.
7. Each pass through the run files provides larger and larger runs until only one run remains.

Ex:

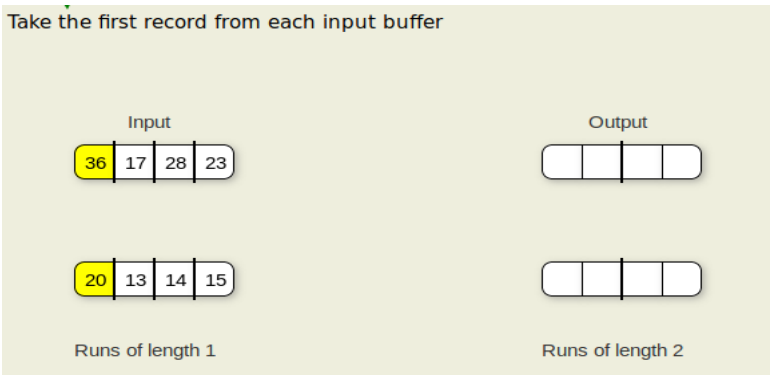
Step 1:



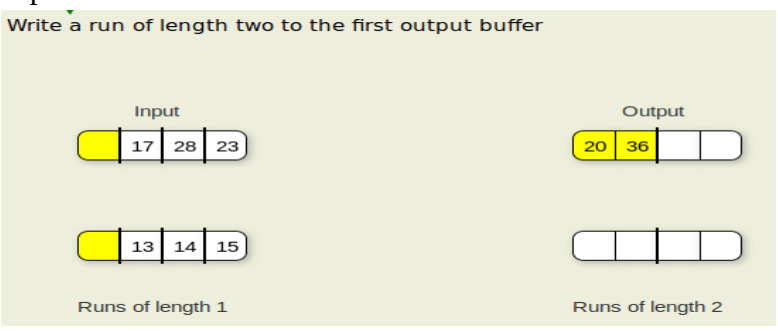
Step 2:



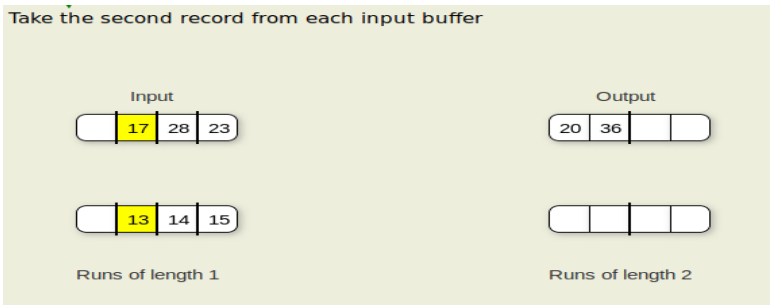
Step 3:



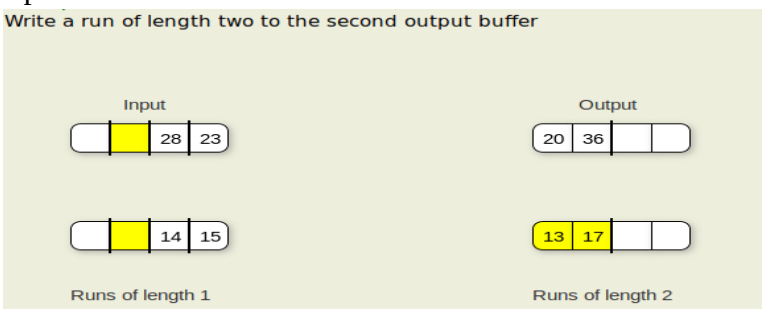
Step 4:



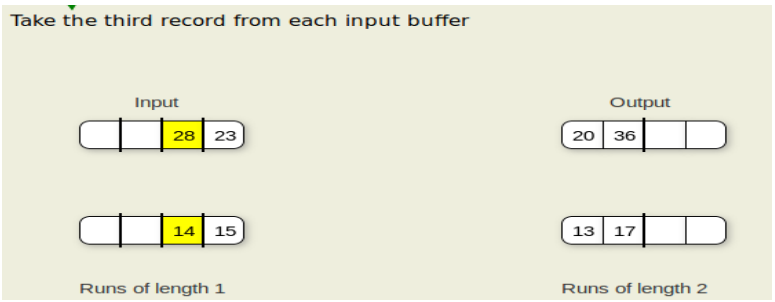
Step 5:



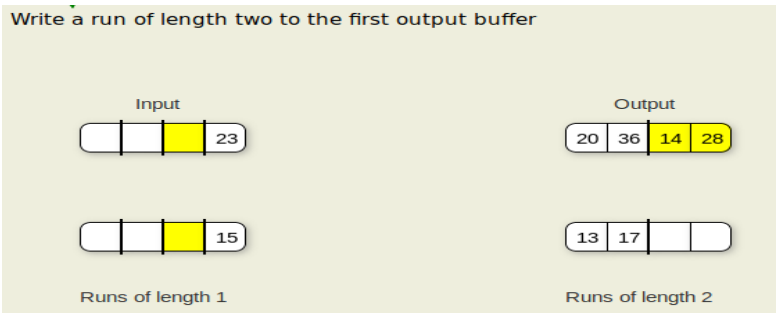
Step 6:



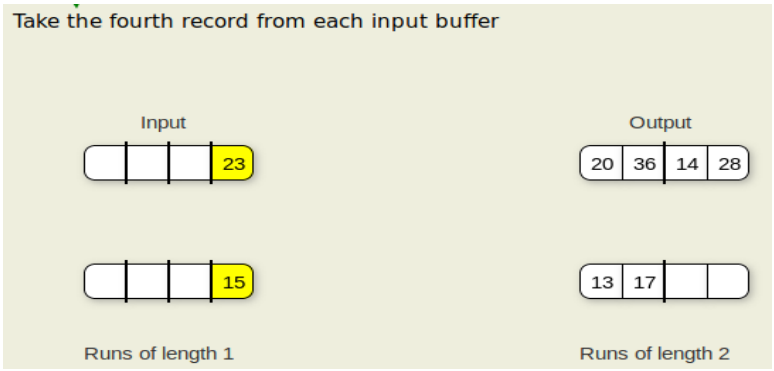
Step 7:



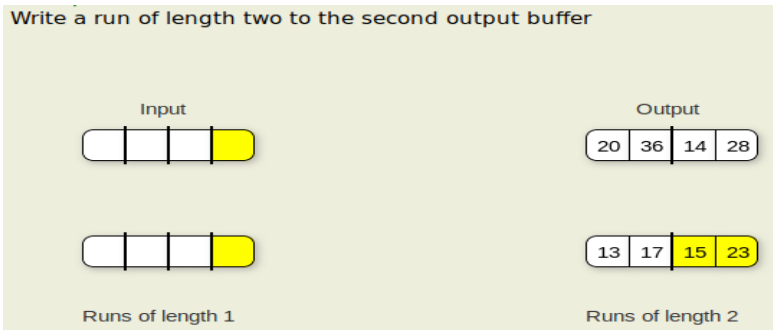
Step 8:



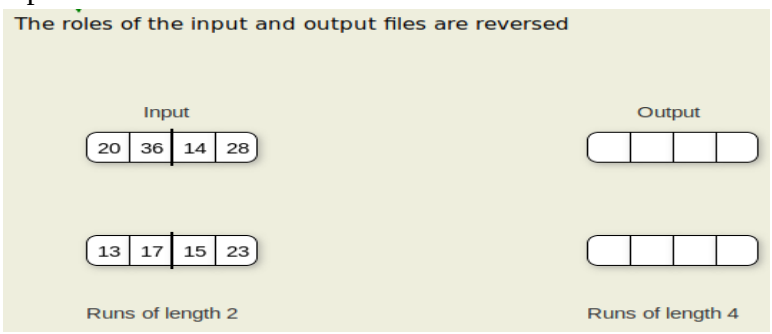
Step 9:



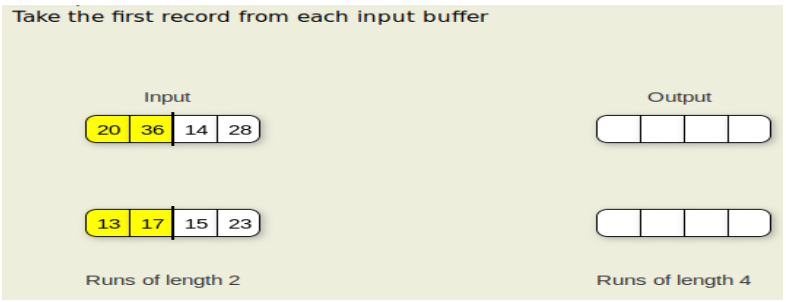
Step 10:



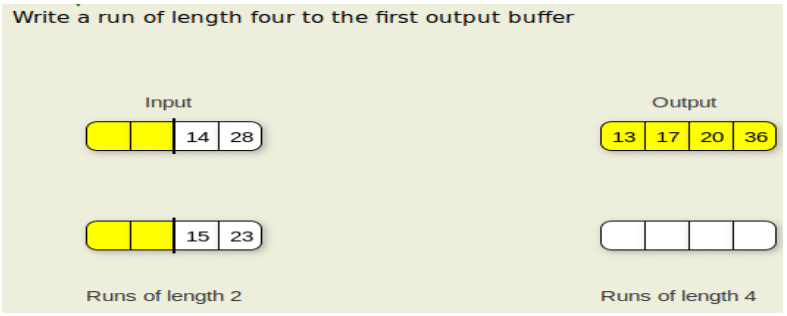
Step 11:



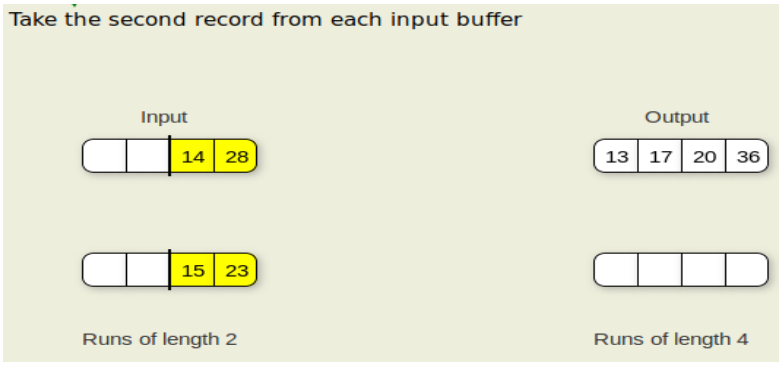
Step 12:



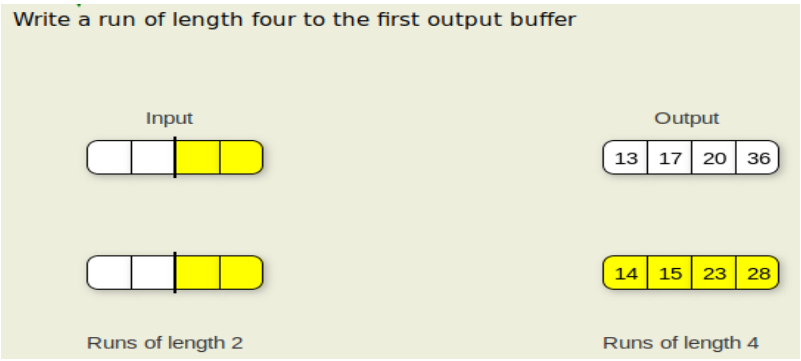
Step 13:



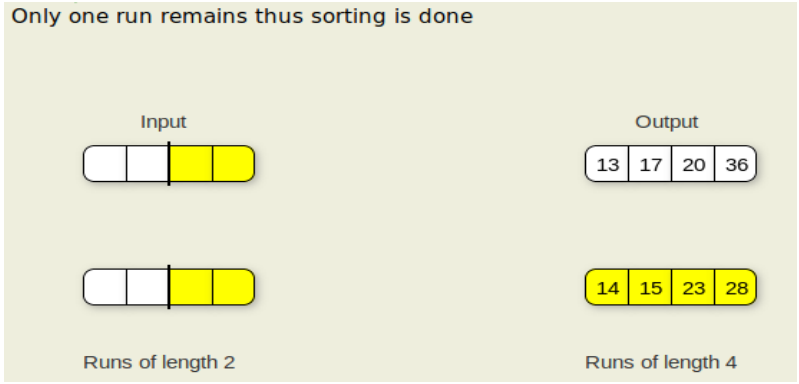
Step 14:



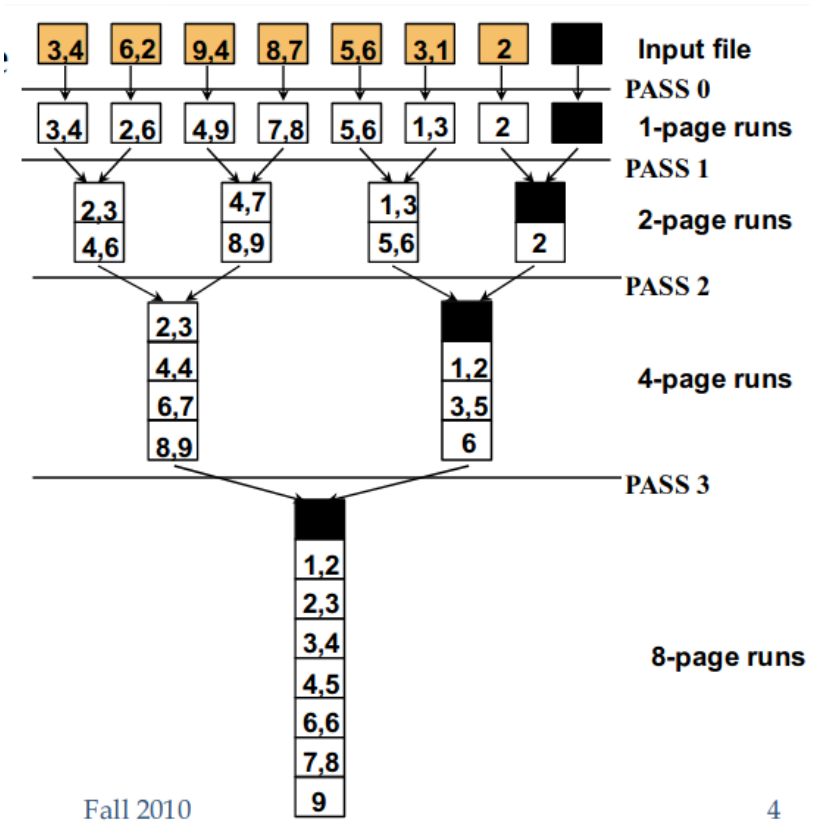
Step 15:



Step 16:



Ex:



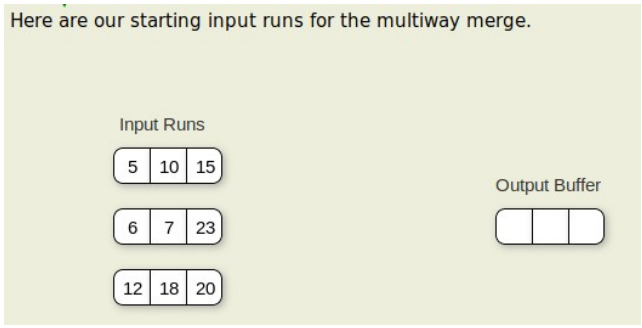
Multiway Merging

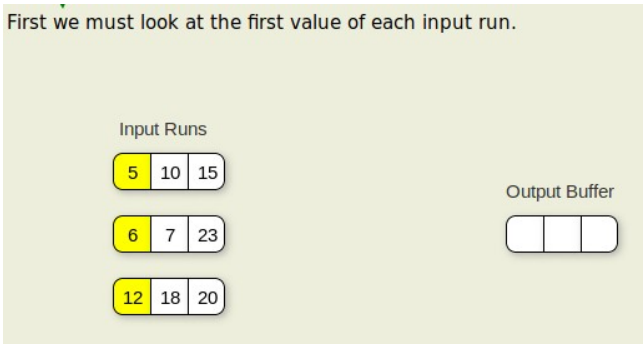
The second stage of a typical external sorting algorithm merges the runs created by the first stage. Assume that we have *R* runs to merge. If a simple two-way merge is used, then *R* runs (regardless of their sizes) will require log*R* passes through the file. While *R* should be much less than the total number of records (because the initial runs should each contain many records), we would like to reduce still further the number of passes required to merge the runs together.

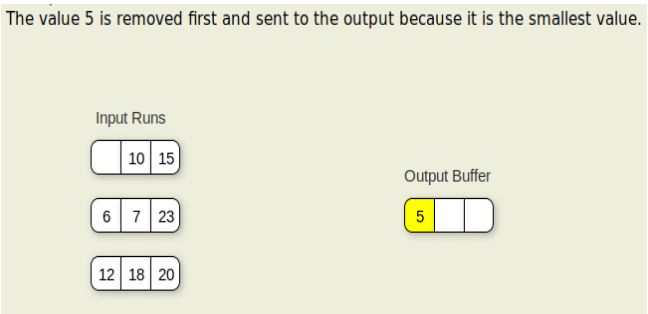
Note that two-way merging does not make good use of available memory. Because merging is a sequential process on the two runs, only one block of records per run need be in memory at a time. Keeping more than one block of a run in memory at any time will not reduce the disk I/O required by the merge process (though if several blocks are read from a file at once time, at least they take advantage of sequential access). Thus, most of the space just used by the heap for replacement selection (typically many blocks in length) is not being used by the merge process.

We can make better use of this space and at the same time greatly reduce the number of passes needed to merge the runs if we merge several runs at a time. Multiway merging is similar to two-way merging. If we have *B* runs to merge, with a block from each run available in memory, then the *B*-way merge algorithm simply looks at *B* values (the front-most value for each input run) and selects the smallest one to output. This value is removed from its run, and the process is repeated. When the current block for any run is exhausted, the next block from that run is read from disk. The following slideshow illustrates a multiway merge.

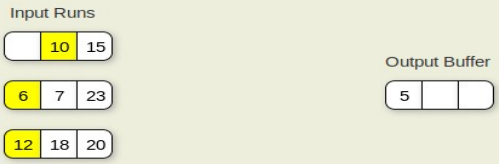
Ex:



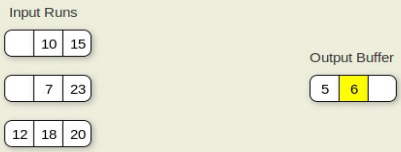




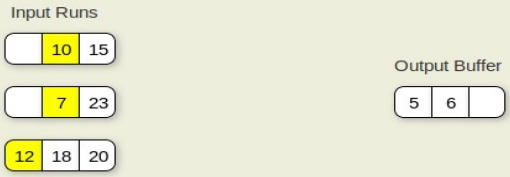
Next we look at the first values in the input runs again.



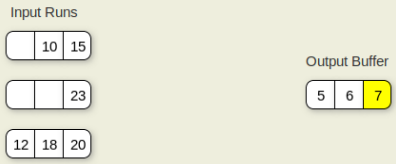
The value 6 is removed first and sent to the output because it is the smallest value.



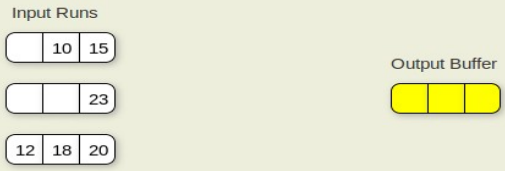
Compare the first values again.



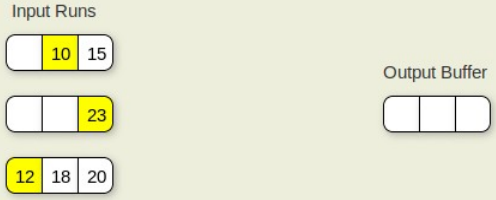
The value 7 is removed first and sent to the output because it is the smallest value.



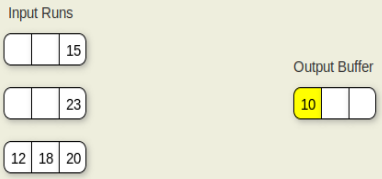
We must write the output buffer to the disk.



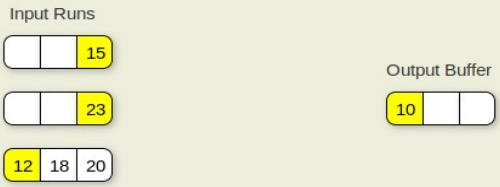
Compare the first values again.



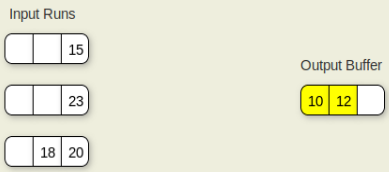
The value 10 is removed first and sent to the output because it is the smallest value.



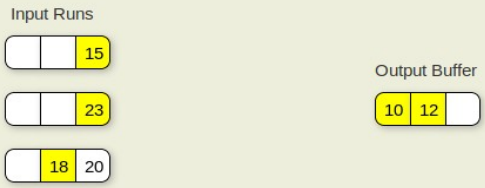
Compare the first values again.



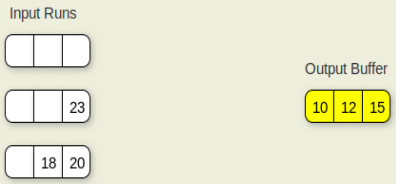
The value 12 is removed first and sent to the output because it is the smallest value.



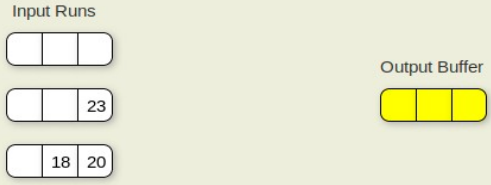
Compare the first values again.



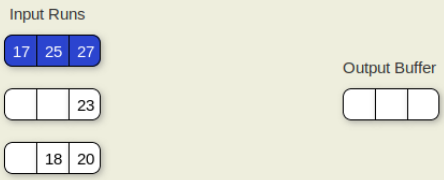
The value 15 is removed first and sent to the output because it is the smallest value.



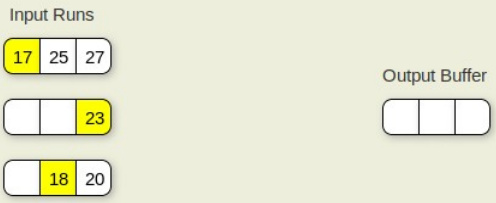
We must write the output buffer to the disk again.



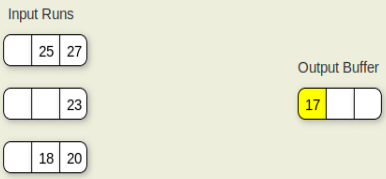
The first run is exhausted. Now we must read in the next block from disk.



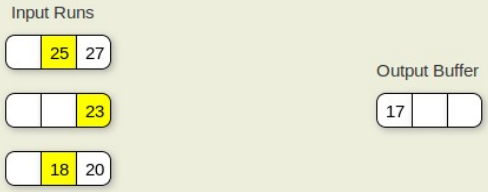
Compare the first values again.



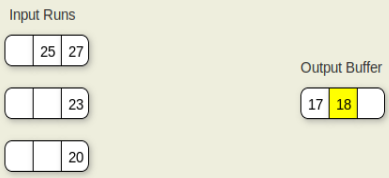
The value 17 is removed first and sent to the output because it is the smallest value.



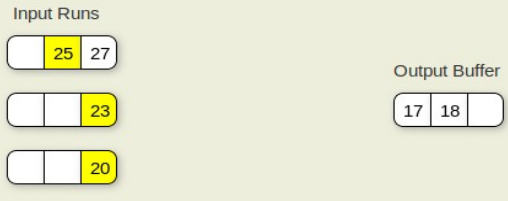
Compare the first values again.



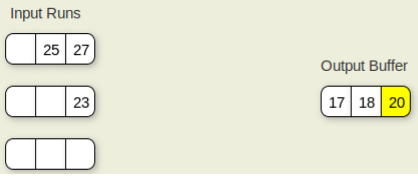
The value 18 is removed first and sent to the output because it is the smallest value.



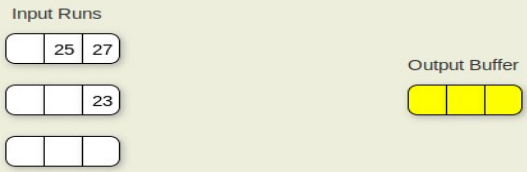
Compare the first values again.



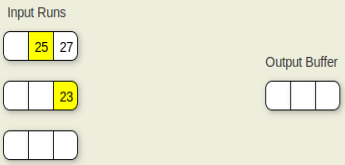
The value 20 is removed first and sent to the output because it is the smallest value.



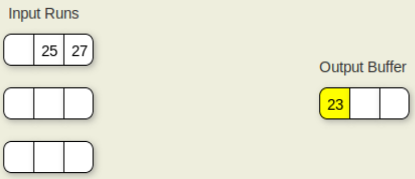
We must write the output buffer to the disk again.



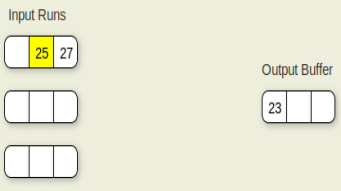
The third run is exhausted but there aren't any more blocks on disk so we compare first values again.



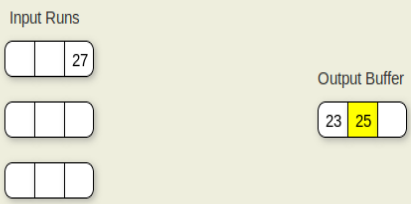
The value 23 is removed first and sent to the output because it is the smallest value.

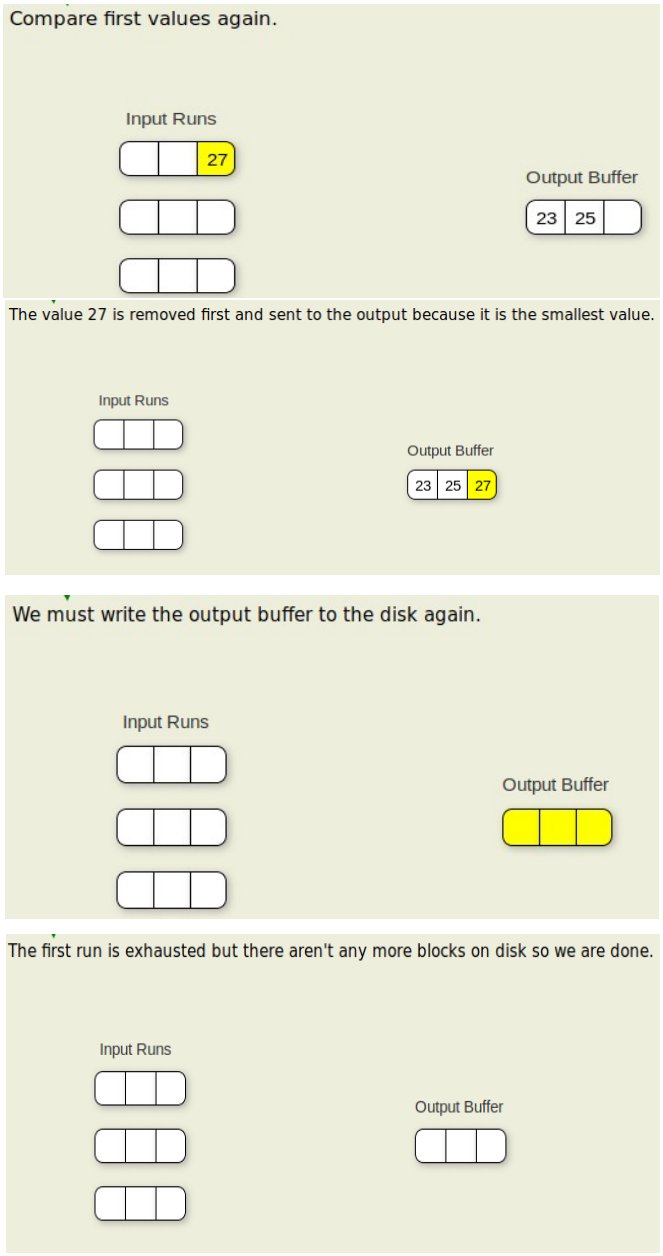


The second run is exhausted but there aren't any more blocks on disk so we compare first values again.



The value 25 is removed first and sent to the output because it is the smallest value.





Conceptually, multiway merge assumes that each run is stored in a separate file. However, this is not necessary in practice. We only need to know the position of each run within a single file, and use `seek` to move to the appropriate block whenever we need new data from a particular run. Naturally, this approach destroys the ability to do sequential processing on the input file. However, if all runs were stored on a single disk drive, then processing would not be truly sequential anyway because the I/O head would be alternating between the runs. Thus, multiway merging replaces several (potentially) sequential passes with a single random access pass. If the processing would not be sequential anyway (such as when all processing is on a single disk drive), no time is lost by doing so.