Breadth First Search

```
#include<iostream>
#include<stdlib.h>
#include<queue>
using namespace std;


class node
{
   public:

    node *left, *right;
    int data;

};

class Breadthfs
{

 public:

 node *insert(node *, int);
 void bfs(node *);

};


node *insert(node *root, int data)
// inserts a node in tree
{

    if(!root)
    {

            root=new node;
            root->left=NULL;
            root->right=NULL;
            root->data=data;
            return root;
    }

    queue<node *> q;
    q.push(root);
```

```
    while(!q.empty())
    {

        node *temp=q.front();
        q.pop();

        if(temp->left==NULL)
        {

            temp->left=new node;
            temp->left->left=NULL;
            temp->left->right=NULL;
            temp->left->data=data;
            return root;
        }
        else
        {

        q.push(temp->left);

        }

        if(temp->right==NULL)
        {

            temp->right=new node;
            temp->right->left=NULL;
            temp->right->right=NULL;
            temp->right->data=data;
            return root;
        }
        else
        {

        q.push(temp->right);

        }

    }

}


void bfs(node *head)
```

```
{

        queue<node*> q;
        q.push(head);

        int qSize;

        while (!q.empty())
        {
                qSize =  q.size();
                #pragma omp parallel for
               //creates parallel threads
                for (int i = 0; i < qSize; i++)
                {
                        node* currNode;
                        #pragma omp critical
                        {
                          currNode = q.front();
                          q.pop();
                          cout<<"\t"<<currNode->data;

                        }// prints parent node
                        #pragma omp critical
                        {
                        if(currNode->left)// push parent's left node in queue
                                q.push(currNode->left);
                        if(currNode->right)
                                q.push(currNode->right);
                        }// push parent's right node in queue

                }
        }

}

int main(){

   node *root=NULL;
   int data;
   char ans;

   do
   {
        cout<<"\n enter data=>";
```

```
        cin>>data;

        root=insert(root,data);

        cout<<"do you want insert one more node?";
        cin>>ans;

    }while(ans=='y'||ans=='Y');

    bfs(root);

    return 0;
}
```

Output:

```
Enter data => 5
Do you want to insert one more node? (y/n) y

Enter data => 3
Do you want to insert one more node? (y/n) y

Enter data => 2
Do you want to insert one more node? (y/n) y

Enter data => 1
Do you want to insert one more node? (y/n) y

Enter data => 7
Do you want to insert one more node? (y/n) y

Enter data => 8
Do you want to insert one more node? (y/n) n

        5       3       7       2       1       8
```

Depth First Search

```
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>

using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
        stack<int> s;
        s.push(node);

        while (!s.empty())  {
        int curr_node = s.top();
        s.pop();

        if (!visited[curr_node]) {
        visited[curr_node] = true;

        #pragma omp parallel for
        for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                s.push(adj_node);
                }
        }
        }
        }
}

int main() {
        int n, m, start_node;
        cin >> n >> m >> start_node;
      //n: node,m:edges

        for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
//u and v: Pair of edges
```

```
        graph[u].push_back(v);
        graph[v].push_back(u);
        }

        #pragma omp parallel for
        for (int i = 0; i < n; i++) {
        visited[i] = false;
        }

        dfs(start_node);

        for (int i = 0; i < n; i++) {
        if (visited[i]) {
        cout << i << " ";
        }
        }

        return 0;
}
```

Output:

Input:

Copy code

```
6 7 0
0 1
0 2
1 3
2 4
2 5
4 5
5 3
```

Output:

Copy code

```
0 1 2 4 5 3
```

Bubble Sort

```
 #include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;

  void bubble(int *, int);
  void swap(int &, int &);

  void bubble(int *a, int n)
  {
      for(  int i = 0; i < n;  i++ )
       {
      int first = i % 2;

      #pragma omp parallel for shared(a,first)
      for(  int j = first;  j < n-1;  j += 2  )
        {
      if(  a[ j ]  >  a[ j+1 ]  )
        {
        swap(  a[ j ],  a[ j+1 ]  );
        }
        }
       }
  }

  void swap(int &a, int &b)
  {

      int test;
      test=a;
      a=b;
      b=test;
```

```
}

int main()
{

    int *a,n;
    cout<<"\n enter total no of elements=>";
    cin>>n;
    a=new int[n];
    cout<<"\n enter elements=>";
    for(int i=0;i<n;i++)
    {
    cin>>a[i];
    }

    bubble(a,n);

    cout<<"\n sorted array is=>";
    for(int i=0;i<n;i++)
    {
    cout<<a[i]<<endl;
    }

return 0;
}
```

Input-

enter total no of elements=>5

enter elements=>3

5

2

4

1

Output-

sorted array is=>1

2

3

4

5

*/

```c
Sorted array: 1 2 3 4 5 6 7 8 9
```

Merge Sort

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

void mergesort(int a[],int i,int j)
{
int mid;
if(i<j)
{
    mid=(i+j)/2;

    #pragma omp parallel sections
    {

       #pragma omp section
       {
          mergesort(a,i,mid);
       }

       #pragma omp section
       {
          mergesort(a,mid+1,j);
       }
    }

    merge(a,i,mid,mid+1,j);
```

```
}

}

void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[1000];
int i,j,k;
i=i1;
j=i2;
k=0;

while(i<=j1 && j<=j2)
{
    if(a[i]<a[j])
    {
        temp[k++]=a[i++];
    }
    else
    {
        temp[k++]=a[j++];
    }
}

while(i<=j1)
{
    temp[k++]=a[i++];
}

while(j<=j2)
{
    temp[k++]=a[j++];
```

```
}

for(i=i1,j=0;i<=j2;i++,j++)
{
    a[i]=temp[j];
}
}

int main()
{
int *a,n,i;
cout<<"\n enter total no of elements=>";
cin>>n;
a= new int[n];
cout<<"\n enter elements=>";
for(i=0;i<n;i++)
{
    cin>>a[i];
}
  // start=.......
//#pragma omp…..
mergesort(a, 0, n-1);
//          stop…….
cout<<"\n sorted array is=>";
for(i=0;i<n;i++)
{
    cout<<"\n"<<a[i];
}
  // Cout<<Stop-Start
return 0;
}
```

Input-

enter total no of elements=>6

enter elements=>3
4
2
6
1
5

Output-
sorted array is=>
1
2
3
4
5
6 */

Min, Max, Sum and Average Operation

```
#include <iostream>
//#include <vector>
#include <omp.h>
#include <climits>
using namespace std;
void min_reduction(int arr[], int n) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < n; i++) {
if (arr[i] < min_value) {
   min_value = arr[i];
}
  }
  cout << "Minimum value: " << min_value << endl;
}

void max_reduction(int arr[], int n) {
  int max_value = INT_MIN;
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < n; i++) {
if (arr[i] > max_value) {
   max_value = arr[i];
}
  }
  cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(int arr[], int n) {
  int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++) {
sum += arr[i];
  }
  cout << "Sum: " << sum << endl;
}
```

```
void average_reduction(int arr[], int n) {
  int sum = 0;
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n; i++) {
sum += arr[i];
  }
  cout << "Average: " << (double)sum / (n-1) << endl;
}

int main() {
    int *arr,n;
    cout<<"\n enter total no of elements=>";
    cin>>n;
    arr=new int[n];
    cout<<"\n enter elements=>";
    for(int i=0;i<n;i++)
    {
    cin>>arr[i];
    }

//  int arr[] = {5, 2, 9, 1, 7, 6, 8, 3, 4};
//  int n = size(arr);

  min_reduction(arr, n);
  max_reduction(arr, n);
  sum_reduction(arr, n);
  average_reduction(arr, n);
}
```

/* Output

enter total no of elements=>5

enter elements=>8
6
3
4
2
Minimum value: 2
Maximum value: 8
Sum: 23
Average: 5.75