```
Define a Node structure for the BST //
} struct Node
int key; // the integer key of the node
Node* left; // pointer to the left child node
Node* right; // pointer to the right child node
;{

Implement the newNode() function to create a new node with a given key //
} Node* newNode(int key)
allocate memory for a new node //
;()Node* node = new Node
assign the key and initialize the child pointers to NULL //
;node->key = key
;node->left = NULL
;node->right = NULL
return the new node //
;return node
{

Implement the insertRec() function to recursively insert a new key into the BST //
} Node* insertRec(Node* root, int key)
base case: if the root is NULL, create a new node with the key and return it //
} if (root == NULL)
;return newNode(key)
{
recursive case: if the key is less than the root's key, insert it in the left subtree //
} if (key < root->key)
;root->left = insertRec(root->left, key)
{
recursive case: if the key is greater than the root's key, insert it in the right subtree //
} else if (key > root->key)
;root->right = insertRec(root->right, key)
{
return the root of the modified tree //
;return root
{
```

```
()Implement the insert() function as a wrapper around insertRec //
} Node* insert(Node* root, int key)
call the insertRec() function with the root and the key and return the new root //
;return insertRec(root, key)
{

Implement the search() function to check if a key exists in the BST //
} bool search(Node* root, int key)
base case: if the root is NULL, the key is not found //
} if (root == NULL)
;return false
{
base case: if the root's key is equal to the key, the key is found //
} if (root->key == key)
;return true
{
recursive case: if the key is less than the root's key, search in the left subtree //
} if (key < root->key)
;return search(root->left, key)
{
recursive case: if the key is greater than the root's key, search in the right subtree //
} else
;return search(root->right, key)
{
{

Graph class for DFS //
} class Graph
number of vertices //
;int numVertices
array of lists for adjacency lists //
;list<int>* adjLists
array of booleans for visited vertices //
;bool* visited
:public
constructor //
;Graph(int V)
```

```
method to add an edge //
;void addEdge(int src, int dest)
method to perform DFS //
;void DFS(int start)
;{

constructor //
} Graph::Graph(int V)
initialize numVertices //
;numVertices = V
create new array of lists //
;adjLists = new list<int>[V]
create new array of booleans //
;visited = new bool[V]
initialize all vertices as not visited //
} for (int i = 0; i < V; i++)
;visited[i] = false
{
{

method to add an edge //
} void Graph::addEdge(int src, int dest)
add dest to the adjacency list of src //
;adjLists[src].push_back(dest)
{

method to perform DFS //
} void Graph::DFS(int start)
mark the start vertex as visited //
;visited[start] = true
print the start vertex //
;" " >> cout << start
iterate over the adjacency list of start //
} for (auto i = adjLists[start].begin(); i != adjLists[start].end(); i++)
if the adjacent vertex is not visited, recursively call DFS on it //
} if (!visited[*i])
;DFS(*i)
```

```
{
{
{

Graph class for BFS //
} class Graph
number of vertices //
;int numVertices
array of lists for adjacency lists //
;list<int>* adjLists
array of booleans for visited vertices //
;bool* visited
:public
constructor //
;Graph(int V)
method to add an edge //
;void addEdge(int src, int dest)
method to perform BFS //
;void BFS(int start)
;{

constructor //
} Graph::Graph(int V)
initialize numVertices //
;numVertices = V
create new array of lists //
;adjLists = new list<int>[V]
create new array of booleans //
;visited = new bool[V]
initialize all vertices as not visited //
} for (int i = 0; i < V; i++)
;visited[i] = false
{
{

method to add an edge //
} void Graph::addEdge(int src, int dest)
```

```
add dest to the adjacency list of src //
;adjLists[src].push_back(dest)
{

method to perform BFS //
} void Graph::BFS(int start)
create a queue for BFS //
;queue<int> q
mark the start vertex as visited and enqueue it //
;visited[start] = true
;q.push(start)
loop until the queue is empty //
} while (!q.empty())
dequeue a vertex from the queue and print it //
;()int v = q.front
;()q.pop
;" " >> cout << v
get all adjacent vertices of the dequeued vertex //
} for (auto i = adjLists[v].begin(); i != adjLists[v].end(); i++)
if an adjacent vertex has not been visited, mark it visited and enqueue it //
} if (!visited[*i])
;visited[*i] = true
;q.push(*i)
{
{
{
{

Define a struct for the max heap //
} struct MaxHeap
array to store the elements //
;int* arr
size of the array //
;int size
capacity of the array //
;int capacity
;{
```

```
Implement a function to create a new max heap with a given capacity //
} MaxHeap* createMaxHeap(int capacity)
allocate memory for a new max heap //
;()MaxHeap* maxHeap = new MaxHeap
initialize the size to zero //
;maxHeap->size = 0
initialize the capacity to the given value //
;maxHeap->capacity = capacity
allocate memory for the array //
;maxHeap->arr = new int[capacity]
return the max heap //
;return maxHeap
{

Implement a function to swap two elements in the array //
} void swap(int* a, int* b)
store the value of a in a temporary variable //
;int temp = *a
assign the value of b to a //
;a = *b*
assign the value of temp to b //
;b = temp*
{

Implement a function to insert an element into the heap //
} void insert(MaxHeap* maxHeap, int x)
check if the heap is full //
} if (maxHeap->size == maxHeap->capacity)
throw an exception //
;throw runtime_error("Heap is full")
{
increment the size of the heap //
;++maxHeap->size
insert the element at the end of the array //
;int i = maxHeap->size - 1
;maxHeap->arr[i] = x
```

```
fix the max heap property if violated //
} while (i != 0 && maxHeap->arr[i] > maxHeap->arr[(i - 1) / 2])
swap the current element with its parent //
;swap(&maxHeap->arr[i], &maxHeap->arr[(i - 1) / 2])
update the index to the parent //
;i = (i - 1) / 2
{
{

Implement a function to heapify a subtree rooted at a given index //
} void heapify(MaxHeap* maxHeap, int i)
get the indices of the left and right children //
;int left = 2 * i + 1
;int right = 2 * i + 2
initialize the largest element as the current element //
;int largest = i
compare the current element with its left child //
} if (left < maxHeap->size && maxHeap->arr[left] > maxHeap->arr[largest])
update the largest element to the left child //
;largest = left
{
compare the current element with its right child //
} if (right < maxHeap->size && maxHeap->arr[right] > maxHeap->arr[largest])
update the largest element to the right child //
;largest = right
{
check if the largest element is not the current element //
} if (largest != i)
swap the current element with the largest element //
;swap(&maxHeap->arr[i], &maxHeap->arr[largest])
recursively heapify the affected subtree //
;heapify(maxHeap, largest)
{
{

Implement a function to remove and return the maximum element from the heap //
} int extractMax(MaxHeap* maxHeap)
```

```
check if the heap is empty //
} if (maxHeap->size == 0)
throw an exception //
;throw runtime_error("Heap is empty")
{
get the maximum element from the root of the heap //
;int max = maxHeap->arr[0]
replace the root element with the last element //
;maxHeap->arr[0] = maxHeap->arr[maxHeap->size - 1]
decrement the size of the heap //
;--maxHeap->size
heapify the root element //
;heapify(maxHeap, 0)
return the maximum element //
;return max
{

Implement a function to return the maximum element without removing it //
} int getMax(MaxHeap* maxHeap)
check if the heap is empty //
} if (maxHeap->size == 0)
throw an exception //
;throw runtime_error("Heap is empty")
{
return the root element of the heap //
;return maxHeap->arr[0]
{
```