UNIT 4 PROGRAMS

```
#include<iostream>
 using namespace std;
class TreeNode
{
        public:
    int value;
        TreeNode * left;
        TreeNode * right;

        TreeNode()
        {
                value = 0;
        left = NULL;
        right = NULL;
        }
        TreeNode(int v)
        {
        value = v;
        left = NULL;
        right = NULL;
        }
};

class BST
{
        public:
    TreeNode * root;
        BST()
        {
```

```
        root = NULL;
        }
        bool isTreeEmpty()
        {
        if (root == NULL)
                {
                return true;
        }
                else
                {
                return false;
        }                                                                    2
  }

void insertNode(TreeNode* new_node)
{
   if (root == NULL)
        {
        root = new_node;
        cout << "Value Inserted as root node!" << endl;
   }
        else
        {
        TreeNode * temp = root;
        while (temp != NULL)
                {
               if (new_node -> value == temp -> value)
                        {
                 cout << "Value Already exist,"<<"Insert another value!" << endl;
                 return;
               }
                        else if ((new_node -> value < temp -> value) && (temp ->
left == NULL))
                        {
                       temp -> left = new_node;
                        cout << "Value Inserted to the left!" << endl;
                        break;
               }
                        else if (new_node -> value < temp -> value)
                        {
                        temp = temp -> left;
               }
```

```
                    else if ((new_node -> value > temp -> value) && (temp -> right == NULL))
                    {
                    temp -> right = new_node;
                    cout << "Value Inserted to the right!" << endl;
                    break;
            }
                    else
                    {
                    temp = temp -> right;
        }
      }                                                                       3
    }
}


void printPreorder(TreeNode * r) //(current node, Left, Right)
{
   if (r == NULL)
     return;
   /* first print data of node */
   cout << r -> value << " ";
   /* then recur on left sutree */
   printPreorder(r -> left);
   /* now recur on right subtree */
   printPreorder(r -> right);
}

void printInorder(TreeNode * r) //  (Left, current node, Right)
{
   if (r == NULL)
     return;
   /* first recur on left child */
   printInorder(r -> left);
   /* then print the data of node */
   cout << r -> value << " ";
   /* now recur on right child */
   printInorder(r -> right);
}
void printPostorder(TreeNode * r) //(Left, Right, Root)
{
   if (r == NULL)
```

```
      return;
    // first recur on left subtree
    printPostorder(r -> left);
    // then recur on right subtree
    printPostorder(r -> right);
    // now deal with the node
    cout << r -> value << " ";
}

TreeNode* Search(TreeNode* r, int val)
{
    if (r == NULL || r -> value == val)                                        4
        return r;
    else if (val < r -> value)
        return Search(r -> left, val);
    else
      return Search(r -> right, val);
  }

int height(TreeNode * r)
{
    if (r == NULL)
      return -1;
    else
        {
      /* compute the height of each subtree */
        int lheight = height(r -> left);
        int rheight = height(r -> right);

      /* use the larger one */
        if (lheight > rheight)
        return (lheight + 1);
        else
                    return (rheight + 1);
    }
}

TreeNode * minValueNode(TreeNode * node)
{
    TreeNode * current = node;
    /* loop down to find the leftmost leaf */
    while (current -> left != NULL)
```

```
        {
        current = current -> left;
    }

    return current;
  }

TreeNode * deleteNode(TreeNode * r, int v)
{
    // base case
    if (r == NULL)
        {                                                                    5
        return NULL;
    }
    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    else if (v < r -> value)
        {
        r -> left = deleteNode(r -> left, v);
    }
    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (v > r -> value)
        {
      r -> right = deleteNode(r -> right, v);
    }
    // if key is same as root's key, then This is the node to be deleted
    else
        {
      // node with only one child or no child
            if (r -> left == NULL)
                {
              TreeNode * temp = r -> right;
              delete r;
              return temp;
            }
                else if (r -> right == NULL)
                {
              TreeNode * temp = r -> left;
              delete r;
              return temp;
            }
```

```
              else
              {
             // node with two children: Get the inorder successor (smallest
             // in the right subtree)
             TreeNode * temp = minValueNode(r -> right);
             // Copy the inorder successor's content to this node
             r -> value = temp -> value;
             // Delete the inorder successor
             r -> right = deleteNode(r -> right, temp -> value);
             //deleteNode(r->right, temp->value);
           }
    }                                                                   6
    return r;
  }

};

int main()
{
     BST obj;
     int option, val;

     do
     {
    cout << "What operation do you want to perform? " <<endl;
    cout << "1. Insert Node" << endl;
    cout << "2. Search Node" << endl;
    cout << "3. Delete Node" << endl;
    cout << "4. Print/Traversal BST values" << endl;
    cout << "5. Height of Tree" << endl;
    cin >> option;
    //Node n1;
    TreeNode * new_node = new TreeNode();

    switch (option)
      {
    case 0:
      break;
    case 1:
      cout <<"INSERT"<<endl;
        cout <<"Enter VALUE of TREE NODE to INSERT in BST: ";
        cin >> val;
```

```
        new_node->value = val;
        obj.insertNode(new_node);
        cout<<endl;
      break;

case 2:
          cout << "SEARCH" << endl;
          cout << "Enter VALUE of TREE NODE to SEARCH in BST: ";
          cin >> val;
          new_node = obj.Search(obj.root, val);
          if (new_node != NULL)
              {                                                    7
           cout << "Value found" << endl;
          } else
              {
           cout << "Value NOT found" << endl;
          }
          break;
case 3:
          cout << "DELETE" << endl;
          cout << "Enter VALUE of TREE NODE to DELETE in BST: ";
          cin >> val;
          new_node = obj.Search(obj.root, val);
          if (new_node != NULL)
              {
           obj.deleteNode(obj.root, val);
           cout << "Value Deleted" << endl;
          }
              else
              {
           cout << "Value NOT found" << endl;
          }
          break;
case 4:
      cout <<"PRE-ORDER: ";
      obj.printPreorder(obj.root);
      cout<<endl;
      cout <<"IN-ORDER: ";
      obj.printInorder(obj.root);
      cout<<endl;
      cout <<"POST-ORDER: ";
      obj.printPostorder(obj.root);
```

```
        cout<<endl;
        break;
   case 5:
           cout << "TREE HEIGHT" << endl;
           cout << "Height : " << obj.height(obj.root) << endl;
           break;
   default:
     cout << "Enter Proper Option number " << endl;
   }

  } while (option != 0);

  return 0;
}
```


MERGING ARRAYS

```
// C++ program to merge two sorted arrays/
#include<iostream>
using namespace std;

void mergeArrays(int arr1[], int arr2[], int n1,
                                        int n2, int arr3[])
{
        int i = 0, j = 0, k = 0;

        // Traverse both array
        while (i<n1 && j <n2)
        {
                // Check if current element of first
                // array is smaller than current element
                // of second array. If yes, store first
                // array element and increment first array
                // index. Otherwise do same with second array
                if (arr1[i] < arr2[j])
                        arr3[k++] = arr1[i++];
                else
                        arr3[k++] = arr2[j++];
        }

        // Store remaining elements of first array
        while (i < n1)
                arr3[k++] = arr1[i++];
```

```
	// Store remaining elements of second array
	while (j < n2)
		arr3[k++] = arr2[j++];
}

int main()
{
	int arr1[] = {1, 3, 5, 7};
	int n1 = sizeof(arr1) / sizeof(arr1[0]);

	int arr2[] = {2, 4, 6, 8};
	int n2 = sizeof(arr2) / sizeof(arr2[0]);

	int arr3[n1+n2];
	mergeArrays(arr1, arr2, n1, n2, arr3);

	cout << "Array after merging" <<endl;
	for (int i=0; i < n1+n2; i++)
		cout << arr3[i] << " ";

	return 0;
}
```

Output

```
1 2 3 4 5 6 7 8
```

Time Complexity: O(N)

Auxiliary Space: O(N)

QUICK SORT ON ARRAYS

```
#include<iostream>
using namespace std;

void swap(int* x,int* y)
{
	int temp;
	temp = *y;
	*y = *x;
	*x = temp;
}
int partition(int arr[],int low,int high)
```

```
{
	int i,j,pivot;
	pivot = arr[low];
	i = low+1;
	j = high;
	while(i<=j)
	{
		while(arr[i]<pivot && i<=high)
			i++;
		while(arr[j]>pivot && j>=low)
			j--;
		if(i<j)
		{
			swap(&arr[i],&arr[j]);
		}
	}
	arr[low] = arr[j];
	arr[j] = pivot;
	return j;
}

void Quick_Sort(int arr[],int low,int high)
{
	int i;
	if(low<high)
	{
		i = partition(arr,low,high);
		Quick_Sort(arr,low,i-1);
		Quick_Sort(arr,i+1,high);
	}
}

void Print_Array(int arr[],int n)
{
	cout<<"Sorted Array is"<<endl;
	for(int i=0;i<n;i++)
	{
		cout<<arr[i]<<" ";
	}
	cout<<endl;
}
```

```
int main()
{
	int n;
	int arr[100];
	cout<<"Enter size of array"<<endl;
	cin>>n;
	cout<<"Enter elements of array"<<endl;
	for(int i=0;i<n;i++)
	{
		cin>>arr[i];
	}
	cout<<"successfulling entered elements"<<endl;
	Quick_Sort(arr,0,n-1);
	Print_Array(arr,n);
	return 0;
}
```

QUICK SORT ON LISTS

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <cstdio>
using namespace std;


struct Node
{
	int data;
	struct Node *next;
};


void push(struct Node** head_ref, int new_data)
{
	/* allocate node */
	struct Node* new_node = new Node;

	/* put in the data */
	new_node->data = new_data;

	/* link the old list off the new node */
	new_node->next = (*head_ref);
```

```
	/* move the head to point to the new node */
	(*head_ref) = new_node;
}

/* A utility function to print linked list */
void printList(struct Node *node)
{
	while (node != NULL)
	{
		cout<<node->data<<" ";
		node = node->next;
	}
	cout<<"\n";
}

// Returns the last node of the list
struct Node *getTail(struct Node *cur)
{
	while (cur != NULL && cur->next != NULL)
		cur = cur->next;
	return cur;
}

// Partitions the list taking the last element as the pivot
struct Node *partition(struct Node *head, struct Node *end,
					struct Node **newHead, struct Node **newEnd)
{
	struct Node *pivot = end;
	struct Node *prev = NULL, *cur = head, *tail = pivot;

	// During partition, both the head and end of the list might change
	// which is updated in the newHead and newEnd variables
	while (cur != pivot)
	{
		if (cur->data < pivot->data)
		{
			// First node that has a value less than the pivot - becomes
			// the new head
			if ((*newHead) == NULL)
				(*newHead) = cur;
```

```
			prev = cur;
			cur = cur->next;
		}
		else // If cur node is greater than pivot
		{
			// Move cur node to next of tail, and change tail
			if (prev)
				prev->next = cur->next;
			struct Node *tmp = cur->next;
			cur->next = NULL;
			tail->next = cur;
			tail = cur;
			cur = tmp;
		}
	}

	// If the pivot data is the smallest element in the current list,
	// pivot becomes the head
	if ((*newHead) == NULL)
		(*newHead) = pivot;

	// Update newEnd to the current last node
	(*newEnd) = tail;

	// Return the pivot node
	return pivot;
}


//here the sorting happens exclusive of the end node
struct Node *quickSortRecur(struct Node *head, struct Node *end)
{
	// base condition
	if (!head || head == end)
		return head;

	Node *newHead = NULL, *newEnd = NULL;

	// Partition the list, newHead and newEnd will be updated
	// by the partition function
	struct Node *pivot = partition(head, end, &newHead, &newEnd);
```

```
        // If pivot is the smallest element - no need to recur for
        // the left part.
        if (newHead != pivot)
        {
                // Set the node before the pivot node as NULL
                struct Node *tmp = newHead;
                while (tmp->next != pivot)
                        tmp = tmp->next;
                tmp->next = NULL;

                // Recur for the list before pivot
                newHead = quickSortRecur(newHead, tmp);

                // Change next of last node of the left half to pivot
                tmp = getTail(newHead);
                tmp->next = pivot;
        }

        // Recur for the list after the pivot element
        pivot->next = quickSortRecur(pivot->next, newEnd);

        return newHead;
}


void quickSort(struct Node **headRef)
{
        (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
        return;
}

int main()
{
        struct Node *a = NULL;
        int n,ele, i=0;

        cout<<"Enter the size of the list"<<endl;
        cin>>n;

        cout<<"Enter elements "<<endl;
        while(i<n)
        {
```

```
		cin>>ele;
		push(&a,ele);
		i++;
	}

	quickSort(&a);

	cout << "Linked List after sorting \n";
	printList(a);

	return 0;
}
```

PRIORITY QUEUE USING LIST

```
// C++ code to implement Priority Queue
// using Linked List
#include <bits/stdc++.h>
using namespace std;

// Node
typedef struct node
{
	int data;

	// Lower values indicate
	// higher priority
	int priority;

	struct node* next;

} Node;

// Function to create a new node
Node* newNode(int d, int p)
{
	Node* temp = (Node*)malloc(sizeof(Node));
	temp->data = d;
	temp->priority = p;
	temp->next = NULL;
```

```
        return temp;
}

// Return the value at head
int peek(Node** head)
{
        return (*head)->data;
}

// Removes the element with the
// highest priority form the list
void pop(Node** head)
{
        Node* temp = *head;
        (*head) = (*head)->next;
        free(temp);
}

// Function to push according to priority
void push(Node** head, int d, int p)
{
        Node* start = (*head);

        // Create new Node
        Node* temp = newNode(d, p);

        // Special Case: The head of list has
        // lesser priority than new node. So
        // insert newnode before head node
        // and change head node.
        if ((*head)->priority > p)
        {

                // Insert New Node before head
                temp->next = *head;
                (*head) = temp;
        }
        else
        {

                // Traverse the list and find a
                // position to insert new node
```

```
        while (start->next != NULL &&
            start->next->priority < p)
        {
            start = start->next;
        }

        // Either at the ends of the list
        // or at required position
        temp->next = start->next;
        start->next = temp;
    }
}

// Function to check is list is empty
int isEmpty(Node** head)
{
    return (*head) == NULL;
}

// Driver code
int main()
{

    // Create a Priority Queue
    // 7->4->5->6
    Node* pq = newNode(4, 1);
    push(&pq, 5, 2);
    push(&pq, 6, 3);
    push(&pq, 7, 0);

    while (!isEmpty(&pq))
    {
        cout << " " << peek(&pq);
        pop(&pq);
    }
    return 0;
}
```

PRIORITY QUEUE USING HEAP

```
// C++ code to implement priority-queue
// using array implementation of
// binary heap

#include <bits/stdc++.h>
using namespace std;

int H[50];
int size = -1;

// Function to return the index of the
// parent node of a given node
int parent(int i)
{

	return (i - 1) / 2;
}

// Function to return the index of the
// left child of the given node
int leftChild(int i)
{

	return ((2 * i) + 1);
}

// Function to return the index of the
// right child of the given node
int rightChild(int i)
{

	return ((2 * i) + 2);
}

// Function to shift up the node in order
// to maintain the heap property
void shiftUp(int i)
{
	while (i > 0 && H[parent(i)] < H[i]) {
```

```
        // Swap parent and current node
        swap(H[parent(i)], H[i]);

        // Update i to parent of i
        i = parent(i);
    }
}

// Function to shift down the node in
// order to maintain the heap property
void shiftDown(int i)
{
    int maxIndex = i;

    // Left Child
    int l = leftChild(i);

    if (l <= size && H[l] > H[maxIndex]) {
        maxIndex = l;
    }

    // Right Child
    int r = rightChild(i);

    if (r <= size && H[r] > H[maxIndex]) {
        maxIndex = r;
    }

    // If i not same as maxIndex
    if (i != maxIndex) {
        swap(H[i], H[maxIndex]);
        shiftDown(maxIndex);
    }
}

// Function to insert a new element
// in the Binary Heap
void insert(int p)
{
    size = size + 1;
    H[size] = p;
```

```
    // Shift Up to maintain heap property
    shiftUp(size);
}

// Function to extract the element with
// maximum priority
int extractMax()
{
    int result = H[0];

    // Replace the value at the root
    // with the last leaf
    H[0] = H[size];
    size = size - 1;

    // Shift down the replaced element
    // to maintain the heap property
    shiftDown(0);
    return result;
}

// Function to change the priority
// of an element
void changePriority(int i, int p)
{
    int oldp = H[i];
    H[i] = p;

    if (p > oldp) {
        shiftUp(i);
    }
    else {
        shiftDown(i);
    }
}

// Function to get value of the current
// maximum element
int getMax()
{
```

```
        return H[0];
}

// Function to remove the element
// located at given index
void remove(int i)
{
        H[i] = getMax() + 1;

        // Shift the node to the root
        // of the heap
        shiftUp(i);

        // Extract the node
        extractMax();
}

// Driver Code
int main()
{

        /*              45
                       /      \
                31      14
                / \ / \
                13 20 7 11
                / \
        12 7
        Create a priority queue shown in
        example in a binary max heap form.
        Queue will be represented in the
        form of array as:
        45 31 14 13 20 7 11 12 7 */

        // Insert the element to the
        // priority queue
        insert(45);
        insert(20);
        insert(14);
        insert(12);
        insert(31);
        insert(7);
```

```
insert(11);
insert(13);
insert(7);

int i = 0;

// Priority queue before extracting max
cout << "Priority Queue : ";
while (i <= size) {
        cout << H[i] << " ";
        i++;
}

cout << "\n";

// Node with maximum priority
cout << "Node with maximum priority : "
        << extractMax() << "\n";

// Priority queue after extracting max
cout << "Priority queue after "
        << "extracting maximum : ";
int j = 0;
while (j <= size) {
        cout << H[j] << " ";
        j++;
}

cout << "\n";

// Change the priority of element
// present at index 2 to 49
changePriority(2, 49);
cout << "Priority queue after "
        << "priority change : ";
int k = 0;
while (k <= size) {
        cout << H[k] << " ";
        k++;
}

cout << "\n";
```

```
	// Remove element at index 3
	remove(3);
	cout << "Priority queue after "
		<< "removing the element : ";
	int l = 0;
	while (l <= size) {
		cout << H[l] << " ";
		l++;
	}
	return 0;
}
```

HEAP SORT

```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
	int largest = i; // Initialize largest as root
	int l = 2 * i + 1; // left = 2*i + 1
	int r = 2 * i + 2; // right = 2*i + 2

	// If left child is larger than root
	if (l < n && arr[l] > arr[largest])
		largest = l;

	// If right child is larger than largest so far
	if (r < n && arr[r] > arr[largest])
		largest = r;

	// If largest is not root
	if (largest != i) {
		swap(arr[i], arr[largest]);

		// Recursively heapify the affected sub-tree
		heapify(arr, n, largest);
```

```
	}
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
	// Build heap (rearrange array)
	for (int i = n / 2 - 1; i >= 0; i--)
		heapify(arr, n, i);

	// One by one extract an element from heap
	for (int i = n - 1; i >= 0; i--) {
		// Move current root to end
		swap(arr[0], arr[i]);

		// call max heapify on the reduced heap
		heapify(arr, i, 0);
	}
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
	for (int i = 0; i < n; ++i)
		cout << arr[i] << " ";
	cout << "\n";
}

// Driver program
int main()
{
	int arr[] = { 12, 11, 13, 5, 6, 7 };
	int n = sizeof(arr) / sizeof(arr[0]);

	heapSort(arr, n);

	cout << "Sorted array is \n";
	printArray(arr, n);
}
```