**Algorithm for conversion of Infix Expresssion to Prefix expression**

Iterate the given expression from left to right, one character at a time

**Step 1:** First reverse the given expression

**Step 2:** If the scanned character is an operand, put it into prefix expression.

**Step 3:** If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

**Step 4:** If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack untill we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.

If the scanned character is opening round bracket ( '(' ), push it into operator's stack.

If the scanned character is closing round bracket ( ')' ), pop out operators from operator's stack until we find an opening bracket ('(' ).

Repeat Step 2,3 and 4 till expression has character

**Step 5:** Now pop out all the remaining operators from the operator's stack and push into postfix expression.

**Step 6:** Exit

**Algorithm for conversion of Infix Expresssion to Postfix expression**

**Step 1:** If the scanned character is an operand, put it into postfix expression.

**Step 2:** If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

**Step 3:** If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack until we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then pop the operators from stack until we find a low precedence operator. If associativity right to left then simply put into stack.

If the scanned character is opening round bracket ( **'('** ), push it into operator's stack.

If the scanned character is closing round bracket ( **')'** ), pop out operators from operator's stack until we find an opening bracket (**'('** ).

Repeat Step 1,2 and 3 till expression has character

**Step 4:** Now pop out all the remaining operators from the operator's stack and push into postfix expression.

**Step 5:** Exit

**Code for Balance Bracket**

#include<iostream>

#include<stack>

using namespace std;

bool isBalanced(string expr) {

stack<char> s;

char ch;

for (int i=0; i<expr.length(); i++) { //for each character in the expression, check conditions

if (expr[i]=='('||expr[i]=='['||expr[i]=='{') { //when it is opening bracket, push into stack

s.push(expr[i]);

continue;

}

if (s.empty()) //stack cannot be empty as it is not opening bracket, there must be closing bracket

return false;

switch (expr[i]) {

case ')': //for closing parenthesis, pop it and check for braces and square brackets

ch = s.top();

s.pop();

if (ch=='{' || ch=='[')

return false;

break;

case '}': //for closing braces, pop it and check for parenthesis and square brackets

ch = s.top();

s.pop();

if (ch=='(' || ch=='[')

return false;

break;

case ']': //for closing square bracket, pop it and check for braces and parenthesis

ch = s.top();

s.pop();

if (ch =='(' || ch == '{')

return false;

break;

}

}

return (s.empty()); //when stack is empty, return true

}

int main() {

string expr = "[{(){()}]";

if (isBalanced(expr))

cout << "Balanced";

else

cout << "Not Balanced";

}

Given “n” ropes of different lengths, connect them into a single rope with minimum cost. Assume that the cost to connect two ropes is the same as the sum of their lengths. (Hint: Use Priority Queue)

**Algorithm**

Begin

   minCost := 0

   fill priority queue with the array elements, (greater value is higher priority)

   while queue is not empty, do

      item1 := get item from queue and delete from queue

      item2 := get item from queue and delete from queue

      minCost := minCost + item1 + item2

      add (item1 + item2) into the queue

   done

   return minCost

End

#include<iostream>

#include<queue>

#include<vector>

using namespace std;

int findMinimumCost(int arr[], int n) {

   //priority queue is set as whose value is bigger, have higher priority

   priority\_queue< int, vector<int>, greater<int>>queue(arr, arr+n);

   int minCost = 0;

   while (queue.size() > 1) {              //when queue has more than one element

      int item1 = queue.top();            //item1 is the shortest element

      queue.pop();

      int item2 = queue.top();          //item2 is bigger than item1 but shorter then other

      queue.pop();

      minCost += item1 + item2;         //connect ropes and add them to the queue

      queue.push(item1 + item2);

   }

   return minCost;

}

int main() {

   int ropeLength[] = {4, 3, 2, 6, 5, 7, 12};

   int n = 7;

   cout << "Total minimum cost: " << findMinimumCost(ropeLength, n);

}

OR

**PseudoCode**

**Algorithm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**procedure** minimiseCost( ):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1.**   Declare min-heap on the lengths of the given ropes.

**2.**Initialise a variable, say **cost=0.**

**3.**   Initialise two variables, **first**and**second.**

**4.**   Pop the top element from the min-heap and assign it to the variable **first**.

**5.**Again, pop the top element from the min-heap and assign it to variable **second.**

**6.**Add the cost of combining these lengths as **cost = cost + first + second**.

**7.**Now **insert this cost of combining these lengths(first + second) into a min-heap**.

**9.**Repeat all the steps mentioned above**till the size of the min-heap becomes less than 2.**

**10.**Return **cost.**

**end procedure**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CODE IN C++**

Now, let’s have a look at the Code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {     //Cost is to find the optimal result to connect all ropes.     //n is for the number of ropes.     //ropeLength is for the length of each rope.     int cost=0,n, ropeLength;     cout<<"Enter the number of ropes";     cin>>n;          //Declaring min-heap     priority\_queue<int, vector<int>, greater<int>>minh;          //Taking input from the user about the rope's length.     for(int i=0; i<n; i++){         cin>>ropeLength;         minh.push(ropeLength);       }          //Run the loop till the size of the minheap becomes less than 2.     while(minh.size()>=2){         int first=minh.top();         minh.pop();         int second=minh.top();         minh.pop();         cost=cost+first+second;         minh.push(first+second);          }          //Returning the result     cout<<cost<<endl; } |

**Output**

|  |
| --- |
| Sample Input:  5  1 2 3 4 5  Sample Output:  33 |

Time Complexity: O(n \*  log n).

**Implement Binary Search Tree**

C++

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

// Create a node

struct node \*newNode(int item) {

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// Inorder Traversal

void inorder(struct node \*root) {

if (root != NULL) {

// Traverse left

inorder(root->left);

// Traverse root

cout << root->key << " -> ";

// Traverse right

inorder(root->right);

}

}

// Insert a node

struct node \*insert(struct node \*node, int key) {

// Return a new node if the tree is empty

if (node == NULL) return newNode(key);

// Traverse to the right place and insert the node

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

// Find the inorder successor

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

// Find the leftmost leaf

while (current && current->left != NULL)

current = current->left;

return current;

}

// Deleting a node

struct node \*deleteNode(struct node \*root, int key) {

// Return if the tree is empty

if (root == NULL) return root;

// Find the node to be deleted

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

// If the node is with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

// If the node has two children

struct node \*temp = minValueNode(root->right);

// Place the inorder successor in position of the node to be deleted

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Driver code

int main() {

struct node \*root = NULL;

root = insert(root, 8);

root = insert(root, 3);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 7);

root = insert(root, 10);

root = insert(root, 14);

root = insert(root, 4);

cout << "Inorder traversal: ";

inorder(root);

cout << "\nAfter deleting 10\n";

root = deleteNode(root, 10);

cout << "Inorder traversal: ";

inorder(root);

}