**Array+Sorts**

*#include <iostream>*

*#include <vector>*

*constexpr std::size\_t SIZE{100};*

*template <typename T>*

*class MyArray*

*{*

*private:*

*T m\_arr[SIZE]{};*

*std::size\_t m\_length{0};*

*void Merge(auto l, auto mid, auto h)*

*{*

*std::vector<T> vec(h - l + 1); // c style array doesnot support dynamc/runtime memory allocation*

*std::size\_t i{l}, j{mid + 1}, length{0};*

*while (i <= mid && j <= h)*

*{*

*if (m\_arr[i] <= m\_arr[j])*

*{*

*vec[length++] = m\_arr[i++];*

*}*

*else*

*{*

*vec[length++] = m\_arr[j++];*

*}*

*}*

*while (i <= mid)*

*{*

*vec[length++] = m\_arr[i++];*

*}*

*while (j <= h)*

*{*

*vec[length++] = m\_arr[j++];*

*}*

*for (std::size\_t index{0}; index < length; index++)*

*{*

*m\_arr[l + index] = vec[index];*

*}*

*}*

*std::size\_t Partition(auto l, auto h)*

*{*

*T pivot{m\_arr[l]};*

*auto i{l}, j{h};*

*while (i < j)*

*{*

*do*

*{*

*i++;*

*} while (i < h && m\_arr[i] <= pivot);*

*do*

*{*

*j--;*

*} while (j > l && m\_arr[j] > pivot);*

*if (i < j)*

*std::swap(m\_arr[i], m\_arr[j]);*

*}*

*std::swap(m\_arr[l], m\_arr[j]);*

*return j;*

*}*

*public:*

*std::size\_t Length() const { return m\_length; }*

*std::size\_t Capacity() const { return SIZE; }*

*void showArray()*

*{*

*std::cout << "The Array is: \n";*

*for (std::size\_t i{0}; i < m\_length; i++)*

*{*

*std::cout << m\_arr[i] << '\n';*

*}*

*}*

*void input(T val)*

*{*

*if (m\_length >= SIZE)*

*{*

*std::cout << "ARRAY ALREADY AT MAX CAPACITY!";*

*return;*

*}*

*m\_arr[m\_length++] = val;*

*}*

*void inputVector(const std::vector<T> &vec)*

*{*

*if (vec.size() > SIZE - m\_length)*

*{*

*std::cout << "Vector too big!\n";*

*return;*

*}*

*for (T val : vec)*

*{*

*m\_arr[m\_length++] = val;*

*}*

*}*

*void insert(T val)*

*{*

*if (m\_length >= SIZE)*

*{*

*std::cout << "ARRAY ALREADY AT MAX CAPACITY!";*

*return;*

*}*

*if (m\_length == 0)*

*{*

*m\_arr[m\_length++] = val;*

*return;*

*}*

*std::size\_t i{m\_length};*

*while(i > 0 && m\_arr[i-1] > val)*

*{*

*m\_arr[i] = m\_arr[i-1];*

*i--;*

*}*

*m\_arr[i] = val;*

*++m\_length;*

*}*

*void bubbleSort()*

*{*

*for (std::size\_t i{0}; i < m\_length - 1; i++)*

*{*

*bool flag{false};*

*for (std::size\_t j{0}; j < m\_length - 1 - i; j++)*

*{*

*if (m\_arr[j + 1] < m\_arr[j])*

*{*

*std::swap(m\_arr[j + 1], m\_arr[j]);*

*flag = true;*

*}*

*}*

*if (!flag)*

*{*

*std::cout << "Array Alr Sorted!\n";*

*return;*

*}*

*}*

*}*

*void selectionSort()*

*{*

*for (std::size\_t i{0}; i < m\_length - 1; i++)*

*{*

*auto min{i};*

*for (auto j{i + 1}; j < m\_length; j++)*

*{*

*if (m\_arr[j] < m\_arr[min])*

*min = j;*

*}*

*std::swap(m\_arr[min], m\_arr[i]);*

*}*

*}*

*void insertionSort()*

*{*

*for(std::size\_t i{1}; i < m\_length; i++)*

*{*

*T key{m\_arr[i]};*

*std::size\_t j{i};*

*while(j > 0 && m\_arr[j-1] > key)*

*{*

*m\_arr[j] = m\_arr[j-1];*

*j--;*

*}*

*m\_arr[j] = key;*

*}*

*}*

*void mergeSort(std::size\_t l = 0, std::size\_t h = SIZE - 1) // default values can only be compile time const*

*{*

*if (h == SIZE - 1)*

*h = m\_length - 1;*

*if (l < h)*

*{*

*auto mid{(h + l) / 2};*

*mergeSort(l, mid);*

*mergeSort(mid + 1, h);*

*Merge(l, mid, h);*

*}*

*}*

*void quickSort(std::size\_t l = 0, std::size\_t h = SIZE)*

*{*

*if (h == SIZE)*

*h = m\_length; // since h is exclusive(h should be infinity)*

*if (l < h)*

*{*

*auto pivot{Partition(l, h)}; //returns index of next pivot*

*quickSort(l, pivot);*

*quickSort(pivot + 1, h);*

*}*

*}*

*};*

*int main()*

*{*

*MyArray<int> a{};*

*a.input(12);*

*a.input(5);*

*a.inputVector({68, 14, 27, 9, 34, 100, 1, 54});*

*a.insertionSort();*

*a.insert(18);*

*a.insert(-2);*

*a.showArray();*

*return 0;*

*}*

**LINEAR QUEUE**

*#include <iostream>*

*constexpr int SIZE{5};*

*class Queue*

*{*

*private:*

*int arr[SIZE];*

*std::size\_t rear{0}, front{0}, count{0};*

*public:*

*bool isFull()*

*{*

*return count == SIZE;*

*}*

*bool isEmpty()*

*{*

*return count == 0;*

*}*

*void enqueue(int val)*

*{*

*if(isFull())*

*{*

*std::cout << "Queue full!";*

*return;*

*}*

*arr[rear++] = val;*

*count++;*

*}*

*void dequeue()*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return;*

*}*

*std::cout << arr[front] << "dequeued" << '\n';*

*front++;*

*count--;*

*}*

*int peek()*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return -1; //////*

*}*

*return(arr[front]);*

*}*

*void display()*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return;*

*}*

*for(std::size\_t i{0}; i < count; i++)*

*{*

*std::cout << arr[front+i] << " ";*

*}*

*std::cout << '\n';*

*}*

*};*

*int main()*

*{*

*Queue q;*

*q.enqueue(54);*

*q.enqueue(2);*

*q.enqueue(12);*

*q.enqueue(33);*

*q.enqueue(23);*

*q.display();*

*q.dequeue();*

*q.display();*

*q.dequeue();*

*q.dequeue();*

*q.dequeue();*

*q.dequeue();*

*q.display();*

*/\* q.enqueue(2); //once linear queue is full, it wont work as intented afte next dequeue cos rear is maxed*

*q.enqueue(12);*

*q.enqueue(33);*

*q.enqueue(23);*

*q.display(); \*/*

*return 0;*

*}*

**CIRCULAR QUEUE**

*#include <iostream>*

*constexpr int SIZE{5};*

*class CircularQueue*

*{*

*private:*

*int arr[SIZE];*

*std::size\_t rear{0}, front{0}, count{0};*

*public:*

*bool isFull() const*

*{*

*return count == SIZE;*

*}*

*bool isEmpty() const*

*{*

*return count == 0;*

*}*

*void enqueue(int val)*

*{*

*if(isFull())*

*{*

*std::cout << "Queue full!";*

*return;*

*}*

*arr[rear] = val;*

*rear = (rear+1)%SIZE; ///change*

*count++;*

*}*

*void dequeue()*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return;*

*}*

*std::cout << arr[front] << " dequeued" << '\n';*

*front = (front+1)%SIZE; //change*

*count--;*

*}*

*int peek() const*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return -1; //////*

*}*

*return(arr[front]);*

*}*

*void display() const*

*{*

*if(isEmpty())*

*{*

*std::cout << "Queue empty!" << '\n';*

*return;*

*}*

*for(std::size\_t i{0}; i < count; i++)*

*{*

*std::cout << arr[(front+i)%SIZE] << " "; ///change*

*}*

*std::cout << '\n';*

*}*

*};*

*int main()*

*{*

*CircularQueue q;*

*q.enqueue(54);*

*q.enqueue(2);*

*q.enqueue(12);*

*q.enqueue(33);*

*q.enqueue(23);*

*q.display();*

*q.dequeue();*

*q.display();*

*q.dequeue();*

*q.dequeue();*

*q.dequeue();*

*q.dequeue();*

*q.display();*

*q.enqueue(2);*

*q.enqueue(12);*

*q.enqueue(33);*

*q.enqueue(23);*

*q.display();*

*return 0;*

*}*

**STACK**

*#include <iostream>*

*constexpr int SIZE{100};*

*template <typename T>*

*class Stack*

*{*

*private:*

*T m\_arr[SIZE]{};*

*std::size\_t m\_top{0}; // alr at first 'index'*

*public:*

*bool isFull()*

*{*

*return m\_top == SIZE;*

*}*

*bool isEmpty()*

*{*

*return m\_top == 0;*

*}*

*void push(T val)*

*{*

*if (isFull())*

*{*

*std::cout << "STACK IS FULL!\n";*

*return;*

*}*

*m\_arr[m\_top++] = val; // stores at 0 and increments to 1*

*}*

*T pop()*

*{*

*if (isEmpty())*

*{*

*std::cout << "stack is empty"; // decrements to 0 then return*

*}*

*return m\_arr[--m\_top];*

*}*

*void show()*

*{*

*for (std::size\_t i{0}; i < m\_top; i++)*

*{*

*std::cout << m\_arr[i] << '\n';*

*}*

*}*

*};*

*int main()*

*{*

*Stack<int> s;*

*s.push(65);*

*s.push(12);*

*s.push(-12);*

*s.push(65);*

*s.push(89);*

*s.show();*

*while (!s.isEmpty())*

*{*

*std::cout << "the popped value is :" << s.pop() << std::endl;*

*}*

*}*

**PARENTHESIS STACK**

*#include <iostream>*

*#include <stack>*

*bool isValid(const std::string &s)*

*{*

*std::stack<char> st{};*

*for (char ch : s)*

*{*

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

*st.push(ch);*

*else*

*{*

*if (st.empty())*

*return false;*

*if ((st.top() == '(' && ch == ')') ||*

*(st.top() == '{' && ch == '}') ||*

*(st.top() == '[' && ch == ']'))*

*st.pop();*

*else*

*return false;*

*}*

*}*

*return st.empty();*

*}*

*int main()*

*{*

*std::string s{"({[]})"};*

*if (isValid(s))*

*std::cout << "valid";*

*else*

*std::cout << "not valid";*

*return 0;*

*}*

**POSTFIX**

*#include <iostream>*

*#include <stack>*

*int charToInt(char ch)*

*{*

*return ch - '0';*

*}*

*bool prcd(char top, char ch)*

*{*

*if((top == '\*' || top == '/' || top == '%') && (ch == '+' || ch == '-'))*

*return true;*

*if((top == '+' || top == '-') && (ch == '+' || ch == '-'))*

*return true;*

*return false;*

*}*

*std::string infixToPost(const std::string& s)*

*{*

*std::stack<char> st{};*

*std::string post{};*

*// std::size\_t i{0};*

*for(char ch: s)*

*{*

*if(std::isdigit(ch))*

*post.push\_back(ch);*

*else*

*{*

*while(!st.empty() && prcd(st.top(), ch))*

*{*

*post.push\_back(st.top());*

*st.pop();*

*}*

*st.push(ch);*

*}*

*}*

*while(!st.empty())*

*{*

*post.push\_back(st.top());*

*st.pop();*

*}*

*std::cout << post << '\n';*

*return post;*

*}*

*void evaluate(const std::string& s)*

*{*

*std::stack<int> st{};*

*for(char ch: s)*

*{*

*if(std::isdigit(ch))*

*st.push(charToInt(ch));*

*else*

*{*

*int oprnd2{st.top()};*

*st.pop();*

*int oprnd1{st.top()};*

*st.pop();*

*switch (ch)*

*{*

*case '+': st.push(oprnd1+oprnd2); break;*

*case '-': st.push(oprnd1-oprnd2); break;*

*case '\*': st.push(oprnd1\*oprnd2); break;*

*case '/': st.push(oprnd1/oprnd2); break;*

*case '%': st.push(oprnd1%oprnd2); break;*

*default: return;*

*}*

*}*

*}*

*std::cout << st.top();*

*}*

*int main()*

*{*

*std::string s{"2+3\*5"};*

*std::string post {infixToPost(s)};*

*evaluate(post);*

*return 0;*

*}*

**PARENTHESIS+POSTFIX**

*#include <iostream>*

*#include <stack>*

*bool prcd(char top, char ch)*

*{*

*if((top == '\*' || top == '/' || top == '%') && (ch == '+' || ch == '-'))*

*return true;*

*if((top == '+' || top == '-') && (ch == '+' || ch == '-'))*

*return true;*

*if((top == '\*' || top == '/' || top == '%') && (ch == '\*' || ch == '/' || ch == '%'))*

*return true;*

*return false;*

*}*

*std::string infixToPost(const std::string& str)*

*{*

*std::stack<char> st{};*

*std::string post{};*

*for(char ch: str)*

*{*

*if(isdigit(ch))*

*post.push\_back(ch);*

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

*st.push(ch);*

*else if(ch == ')' || ch == ']')*

*{*

*while(!st.empty() && ((ch == ')' && st.top() != '(') || (ch == ']' && st.top() != '[')))*

*{*

*post.push\_back(st.top());*

*st.pop();*

*}*

*st.pop();*

*}*

*else*

*{*

*while(!st.empty() && prcd(st.top(), ch) && (st.top() != '(' || st.top() != '['))*

*{*

*post.push\_back(st.top());*

*st.pop();*

*}*

*st.push(ch);*

*}*

*}*

*while(!st.empty())*

*{*

*post.push\_back(st.top());*

*st.pop();*

*}*

*std::cout << post << '\n';*

*return post;*

*}*

*int charToInt(char ch)*

*{*

*return ch - '0';*

*}*

*void evaluate(const std::string& str)*

*{*

*std::stack<int> st{};*

*for(char ch: str)*

*{*

*if(isdigit(ch))*

*st.push(charToInt(ch));*

*else*

*{*

*int oprnd2{st.top()};*

*st.pop();*

*int oprnd1{st.top()};*

*st.pop();*

*switch (ch)*

*{*

*case '+': st.push(oprnd1+oprnd2); break;*

*case '-': st.push(oprnd1-oprnd2); break;*

*case '\*': st.push(oprnd1\*oprnd2); break;*

*case '/': st.push(oprnd1/oprnd2); break;*

*case '%': st.push(oprnd1%oprnd2); break;*

*default: return;*

*}*

*}*

*}*

*std::cout << '\n' << st.top();*

*}*

*int main()*

*{*

*auto post{infixToPost("[(2+3)\*4]/4")};*

*evaluate(post);*

*return 0;*

*}*

**RECURSION**

#include <iostream>

#include <climits>

#include <math.h>

template <typename T>

T fact(T x)

{

if(x == 0)

return 1;

return x\* fact(x-1);

}

template <typename T>

T mult(T x, T y)

{

if(y == 0 || x == 0)

return 0;

if(y < 0)

return -mult(x, -y);

if(y == 1)

return x;

return x+ mult(x, y-1);

}

unsigned int fib(unsigned int x)

{

if(x <= 1)

return x;

return fib(x-1) + fib(x-2);

}

template <typename T>

int binS(T arr[], int low, int high, T key)

{

if(high < low)

return -1;

int mid {(high+low)/2};

if(key == arr[mid])

return mid;

else if(key > arr[mid])

return binS(arr, mid+1, high, key);

else

return binS(arr, low, mid-1, key);

}

bool isSorted(int arr[], std::size\_t i, std::size\_t size)

{

if(i >= size-1)

return true;

if(arr[i] > arr[i+ 1])

return false;

return isSorted(arr, i+1, size);

}

void printAscend(int n)

{

if(n < 0)

return;

printAscend(n-1);

std::cout << n << ' ';

}

int power(int x, int y)

{

if(y == 0)

return 1;

return x\* power(x, y-1);

}

bool isPrime(int x, int i)

{

if(i == 1)

return true;

if(x % i == 0)

return false;

return isPrime(x, i-1);

}

int gcd(int a, int b)

{

if(b == 0)

return a;

return gcd(b, a%b);

}

int sum(int arr[], std::size\_t i)

{

if(i == 0)

return arr[i];

return arr[i] + sum(arr, i-1);

}

const int size{5};

int maxf(int arr[], std::size\_t i)

{

if(i == 0)

return arr[i];

int maxP{maxf(arr, i-1)};

return ((arr[i] > maxP)? arr[i]: maxP);

}

int minf(int arr[], std::size\_t i)

{

if(i == 0)

return arr[i];

int minP{minf(arr, i-1)};

return ((arr[i] < minP)? arr[i]: minP);

}

int secMaxf(int arr[], std::size\_t i, int max, int secMax)

{

if(i == size)

return secMax;

if(arr[i] > max)

{

secMax = max;

max = arr[i];

}

else if(arr[i] > secMax && arr[i] != max)

{

secMax = arr[i];

}

return secMaxf(arr, i+1, max, secMax);

}

int secMinf(int arr[], std::size\_t i, int min, int secMin)

{

if(i == size)

return secMin;

if(arr[i] < min)

{

secMin = min;

min = arr[i];

}

else if(arr[i] < secMin && arr[i] != min)

secMin = arr[i];

return secMinf(arr, i+1, min, secMin);

}

void revString(const char str[])

{

if(\*str != '\0')

{

revString(str+1);

std::cout << \*str;

}

//void returns implicitly

}

void revStdString(std::string& str, std::size\_t begin, std::size\_t end)

{

if(begin >= end)

return;

std::swap(str[begin], str[end]);

revStdString(str, begin+1, end-1);

}

int main()

{

// std::cout << fact<long double>(120);

// std::cout << mult(-2, -2);

// std::cout << fib(3);

/\* int arr[5] {12, 34, 56, 78, 89};

int index {binS(arr, 0, 4, 1)};

if(index == (-1))

std::cout << "not found";

else

{

std::cout << index;

} \*/

// std:: cout << std::boolalpha << isPrime(224, std::sqrt(224));

//std::cout << gcd(4, 8);

/\* const int size{5};

int arr[size] {2, 7, 3, 3, 1};

std:: cout << sum(arr, size-1); \*/

/\* int arr[size] {19, 7, 3, -4, 1};

int max{maxf(arr, size-1)};

std::cout << max;

int min{minf(arr, size-1)};

std::cout << min;

int secMax{};

secMax = secMaxf(arr, 0, INT\_MIN, secMax);

std::cout << secMax;

int secMin{};

secMin = secMinf(arr, 0, INT\_MAX, secMin);

std::cout << secMin; \*/

/\* revString("hello");

std::cout << '\n';

std::string str{"hello"};

revStdString(str, 0, str.size()-1);

std::cout << str; \*/

/\* int arr[5] {1, 45, 78, 90, 567};

std::cout << std::boolalpha << isSorted(arr, 0, 5); \*/

//printAscend(10);

std::cout << power(1, 4);

return 0;

}

**SINGLE LIST**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* next{NULL};*

*};*

*class singleList*

*{*

*private:*

*node\* head{NULL};*

*bool empty() {return head == NULL;}*

*public:*

*node\* search(int val)*

*{*

*node\* temp{head};*

*while(temp != NULL)*

*{*

*if(temp->data == val)*

*return temp;*

*temp = temp->next;*

*}*

*return NULL;*

*}*

*void insertAtHead(int val)*

*{*

*node\* newer = new node;*

*newer->data = val;*

*newer->next = head;*

*head = newer;*

*}*

*void insertAtEnd(int val)*

*{*

*if(empty())*

*{*

*insertAtHead(val);*

*return;*

*}*

*node\* temp{head};*

*while(temp->next != NULL)*

*{*

*temp = temp->next;*

*}*

*node\* newer{new node};*

*newer->data = val;*

*newer->next = NULL;*

*temp->next = newer;*

*}*

*void insertAfter(int val, int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*/\* else if(found->next == NULL) //no need*

*insertAtEnd(val); \*/*

*else*

*{*

*node\* newer{new node};*

*newer->data = val;*

*newer->next = found->next;*

*found->next = newer;*

*}*

*}*

*void insertBefore(int val, int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*else if(found == head)*

*insertAtHead(val);*

*else*

*{*

*node\* newer{new node};*

*newer->data = val;*

*newer->next = found;*

*node\* temp{head};*

*while(temp->next != found)*

*{*

*temp = temp->next;*

*}*

*temp->next = newer;*

*}*

*}*

*void deleteAtHead()*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* temp = head;*

*head = temp->next;*

*std::cout << "deleted value: " << temp->data << '\n';*

*delete temp;*

*}*

*void deleteAtEnd()*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*if(head->next == NULL)*

*{*

*deleteAtHead();*

*return;*

*}*

*node\* temp1{head};*

*node\* temp2{NULL};*

*while(temp1->next != NULL)*

*{*

*temp2 = temp1;*

*temp1 = temp1->next;*

*}*

*temp2->next = NULL;*

*std::cout << "deleted value: "<< temp1->data << '\n';*

*delete temp1;*

*}*

*void deleteAfter(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*else if(found->next == NULL)*

*{*

*std::cout << "nothing to del\n";*

*return;*

*}*

*/\* else if(found->next->next == NULL) //no need*

*deleteAtEnd(); \*/*

*node\* temp{found->next};*

*found->next = temp->next;*

*delete temp;*

*}*

*void deleteBefore(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head)*

*{*

*std::cout << "Nothing to del\n";*

*return;*

*}*

*if(found == head->next)*

*{*

*deleteAtHead();*

*return;*

*}*

*node\* temp{head};*

*while(temp->next->next != found)*

*temp = temp->next;*

*node\* temp2 = temp->next;*

*temp->next = found;*

*delete temp2;*

*}*

*void deleteParticular(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head)*

*deleteAtHead();*

*else if(found->next == NULL)*

*deleteAtEnd();*

*else*

*{*

*node\* temp{head};*

*while(temp->next != found)*

*temp = temp->next;*

*temp->next = found->next;*

*delete found;*

*}*

*}*

*void deleteAllBefore(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head)*

*{*

*std::cout << "nothing to del\n";*

*return;*

*}*

*/\* node\* temp{head};*

*while(temp != found)*

*{*

*int data{temp->data};*

*temp = temp->next;*

*deleteParticular(data);*

*} \*/*

*/\* node\* temp{head};*

*while(temp != found)*

*{*

*temp = temp->next;*

*deleteAtHead();*

*} \*/*

*while(head != found)*

*{*

*node\* toDel{head};*

*head = head->next;*

*delete toDel;*

*}*

*}*

*void deleteAllAfter(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*node\* temp{found->next};*

*while(temp != NULL)*

*{*

*node\* temp2{temp};*

*temp = temp->next;*

*delete temp2;*

*}*

*found->next = NULL;*

*}*

*void delAllOccur(int val)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* temp{head};*

*node\* prev{NULL};*

*while(temp!=NULL)*

*{*

*if(temp->data == val)*

*{*

*if(temp == head)*

*{*

*deleteAtHead();*

*temp = head;*

*}*

*else*

*{*

*prev->next = temp->next;*

*delete temp;*

*temp = prev->next;*

*}*

*}*

*else*

*{*

*prev = temp;*

*temp = temp->next;*

*}*

*}*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node\* temp{head};*

*while(temp != NULL)*

*{*

*std::cout << temp->data << '\n';*

*temp = temp->next;*

*}*

*std::cout << '\n';*

*}*

*void showR(node\* temp)*

*{*

*if(temp == nullptr)*

*return;*

*std::cout << temp->data << " ";*

*showR(temp->next);*

*}*

*void showR() {showR(head);}*

*int sum()*

*{*

*node\* temp{head};*

*int sum{0};*

*while(temp != NULL)*

*{*

*sum += temp->data;*

*temp = temp->next;*

*}*

*return sum;*

*}*

*singleList sum(const singleList& l)*

*{*

*node\* temp{head};*

*node\* temp2{l.head};*

*singleList list{};*

*while(temp != NULL && temp2 != NULL)*

*{*

*list.insertAtEnd(temp->data + temp2->data);*

*temp = temp->next;*

*temp2 = temp2->next;*

*}*

*while(temp != NULL)*

*{*

*list.insertAtEnd(temp->data);*

*temp = temp->next;*

*}*

*while(temp2 != NULL)*

*{*

*list.insertAtEnd(temp2->data);*

*temp2 = temp2->next;*

*}*

*return list;*

*}*

*singleList merge(const singleList& l)*

*{*

*node\* temp{head};*

*node\* temp2{l.head};*

*singleList list{};*

*while(temp != NULL && temp2 != NULL)*

*{*

*if(temp->data <= temp2->data)*

*{*

*list.insertAtEnd(temp->data);*

*temp = temp->next;*

*}*

*else*

*{*

*list.insertAtEnd(temp2->data);*

*temp2 = temp2->next;*

*}*

*}*

*while(temp != NULL)*

*{*

*list.insertAtEnd(temp->data);*

*temp = temp->next;*

*}*

*while(temp2 != NULL)*

*{*

*list.insertAtEnd(temp2->data);*

*temp2 = temp2->next;*

*}*

*return list;*

*}*

*singleList partition()*

*{*

*node\* pivot{head};*

*singleList list1{};*

*while(pivot != NULL)*

*{*

*if(pivot->data % 2 == 0) //pivot->data % 2 is true when there is a remainder*

*list1.insertAtHead(pivot->data);*

*else*

*list1.insertAtEnd(pivot->data);*

*pivot = pivot->next;*

*}*

*return list1;*

*}*

*void reverseData()*

*{*

*node\* start{head};*

*node\* end{head};*

*while(end->next != nullptr)*

*end = end->next;*

*while(start != end)*

*{*

*std::swap(start->data, end->data);*

*start = start->next;*

*node\* temp{head};*

*while(temp->next != end)*

*temp = temp->next;*

*end = temp;*

*}*

*}*

*void reverse()*

*{*

*node\* temp1{nullptr}, \* temp2{head}, \* temp3{head->next};*

*while(temp2 != nullptr)*

*{*

*temp2->next = temp1;*

*temp1 = temp2;*

*temp2 = temp3;*

*if(temp3 != nullptr)*

*temp3 = temp3->next;*

*}*

*head = temp1;*

*}*

*void reverseRecursion(node\* temp, node\* temp2)*

*{*

*if(temp2 == nullptr)*

*{*

*head = temp;*

*return;*

*}*

*node\* temp3{temp2->next};*

*temp2->next = temp;*

*reverseRecursion(temp2, temp3);*

*}*

*};*

*int main()*

*{*

*singleList l1{};*

*l1.insertAtHead(1);*

*l1.insertAtEnd(51);*

*l1.insertAtHead(67);*

*l1.showR();*

*/\* singleList l2{};*

*l2.insertAtEnd(34);*

*l2.insertAtEnd(89);*

*singleList l3{l1.merge(l2)};*

*l3.show(); \*/*

*return 0;*

*}*

**SL QUEUE**

*#include <iostream>*

*template <typename T>*

*struct node*

*{*

*T data{};*

*node\* next{NULL};*

*};*

*template <typename T>*

*class queue*

*{*

*private:*

*node<T>\* head{NULL};*

*node<T>\* tail{NULL};*

*public:*

*bool empty() {return head == NULL;}*

*void push(T val)*

*{*

*node<T>\* newer{new node<T>};*

*newer->data = val;*

*newer->next = NULL;*

*if(empty())*

*head = tail = newer;*

*else*

*{*

*tail->next = newer;*

*tail = newer;*

*}*

*}*

*void pop()*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node<T>\* temp{head};*

*head = head->next;*

*if(empty())*

*tail = NULL;*

*delete temp;*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node<T>\* temp{head};*

*while(temp != NULL)*

*{*

*std::cout << temp->data << '\n';*

*temp = temp->next;*

*}*

*std::cout << '\n';*

*}*

*T front()*

*{*

*return head->data;*

*}*

*std::size\_t size()*

*{*

*std::size\_t size{};*

*node<T>\* temp{head};*

*while(temp != NULL)*

*{*

*size++;*

*temp = temp->next;*

*}*

*return size;*

*}*

*};*

*int main()*

*{*

*queue<int> s{};*

*s.push(5);*

*s.push(3);*

*s.pop();*

*s.push(56);*

*s.push(9);*

*s.show();*

*std::cout << s.front();*

*std::cout << s.size();*

*return 0;*

*}*

**SL STACK**

*#include <iostream>*

*template <typename T>*

*struct node*

*{*

*T data{};*

*node\* next{NULL};*

*};*

*template <typename T>*

*class stack*

*{*

*private:*

*node<T>\* head{NULL};*

*public:*

*bool empty() {return head == NULL;}*

*void push(T val)*

*{*

*node<T>\* newer{new node<T>};*

*newer->data = val;*

*newer->next = head;*

*head = newer;*

*}*

*void pop()*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node<T>\* temp{head};*

*head = head->next;*

*delete temp;*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node<T>\* temp{head};*

*while(temp != NULL)*

*{*

*std::cout << temp->data << '\n';*

*temp = temp->next;*

*}*

*std::cout << '\n';*

*}*

*T top()*

*{*

*return head->data;*

*}*

*std::size\_t size()*

*{*

*std::size\_t size{};*

*node<T>\* temp{head};*

*while(temp != NULL)*

*{*

*size++;*

*temp = temp->next;*

*}*

*return size;*

*}*

*};*

*int main()*

*{*

*stack<int> s{};*

*s.push(5);*

*s.push(3);*

*s.pop();*

*s.push(56);*

*s.push(9);*

*std::cout << s.top();*

*std::cout << s.size();*

*return 0;*

*}*

**DOUBLEE**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* right{nullptr};*

*node\* left{nullptr};*

*};*

*class doublee*

*{*

*private:*

*node\* head{nullptr};*

*public:*

*bool empty() {return head == nullptr;}*

*node\* search(int key)*

*{*

*node\* temp{head};*

*while(temp != nullptr)*

*{*

*if(temp->data == key)*

*return temp;*

*temp = temp->right;*

*}*

*return nullptr;*

*}*

*void insertAtHead(int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(!empty()) //if empty right and left of newer should be nulled which is handled by default initialization*

*{*

*newer->right = head;*

*head->left = newer;*

*}*

*head = newer;*

*}*

*void insertAtEnd(int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*head = newer;*

*return;*

*}*

*node\* temp{head};*

*while(temp->right != nullptr)*

*temp = temp->right;*

*temp->right = newer; //newer right is nullptr by default*

*newer->left = temp;*

*}*

*void insertAfter(int key, int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*head = newer;*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*delete newer; //if key not found remove the node we created*

*return; // then return*

*}*

*newer->left = found;*

*newer->right = found->right;*

*if(found->right != nullptr) //if found isnt at end*

*found->right->left = newer; //update the left addr of found->right*

*found->right = newer; //then update found->right*

*}*

*void insertBefore(int key, int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*head = newer;*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*delete newer;*

*return;*

*}*

*newer->left = found->left;*

*newer->right = found;*

*if(found == head) //if inserting at head*

*head = newer; //update head*

*else //if not*

*found->left->right = newer; //update the prev node*

*found->left = newer;*

*}*

*void delAtHead()*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node\* temp{head->right};*

*delete head;*

*if(temp != nullptr) //if multi nodes,*

*temp->left = nullptr; //update left of next node(new head)*

*head = temp; //update head(incase of single node temp would be null)*

*}*

*void delAtEnd()*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*if(head->right == nullptr) //if single node*

*{*

*delete head;*

*head = nullptr;*

*return;*

*}*

*node\* temp{head};*

*while(temp->right != nullptr)*

*temp = temp->right;*

*temp->left->right = nullptr;*

*delete temp;*

*}*

*/\* void delAtEnd()*

*{*

*if(empty())*

*return;*

*node\* temp{head};*

*while(temp->right != nullptr)*

*temp = temp->right;*

*if(temp == head)*

*head = nullptr;*

*else*

*temp->left->right = nullptr;*

*delete temp;*

*} \*/*

*void delAfter(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*return;*

*}*

*if(found->right == nullptr) //if found is last node*

*std::cout << "nothing to del\n";*

*else*

*{*

*node\* temp{found->right};*

*found->right = temp->right;*

*if(temp->right != nullptr) //if temp(to del) isnt last node*

*temp->right->left = found; //update the left of node after temp*

*delete temp;*

*}*

*}*

*void delBefore(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*return;*

*}*

*if(found == head)*

*{*

*std::cout << "nothing to del\n";*

*return;*

*}*

*node\* temp{found->left};*

*if(temp == head)*

*head = found;*

*else*

*temp->left->right = found;*

*found->left = temp->left;*

*delete temp;*

*}*

*void delParticular(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*return;*

*}*

*if(found == head)*

*delAtHead();*

*else if(found->right == nullptr)*

*delAtEnd();*

*else*

*{*

*found->left->right = found->right;*

*found->right->left = found->left;*

*delete found;*

*}*

*}*

*/\* void deleteParticular(int key)*

*{*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*return;*

*}*

*if(found == head)*

*head = found->right;*

*else*

*found->left->right = found->right;*

*if(found->right != nullptr)*

*found->right->left = found->left;*

*delete found;*

*} \*/*

*void reverse()*

*{*

*node\* prev{nullptr}, \* curr{head};*

*while(curr != nullptr)*

*{*

*curr->left = curr->right;*

*curr->right = prev;*

*prev = curr;*

*curr = curr->left;*

*}*

*head = prev;*

*}*

*/\* void reverse()*

*{*

*node\* temp{nullptr};*

*node\* temp2{head};*

*while(temp2 != nullptr)*

*{*

*temp = temp2->left;*

*temp2->left = temp2->right;*

*temp2->right = temp;*

*temp2 = temp2->left;*

*}*

*if(temp != nullptr)*

*head = temp;*

*} \*/*

*void reverseRecursion(node\* temp2)*

*{*

*if(temp2 == nullptr) //if empty list(nullptr temp2 is passed)*

*return;*

*node\* temp{temp2->left};*

*temp2->left = temp2->right;*

*temp2->right = temp;*

*if(temp2->left == nullptr) //base case: if this is last, make it new head.*

*{*

*head = temp2;*

*return;*

*}*

*reverseRecursion(temp2->left);*

*}*

*void display()*

*{*

*node\* temp = head;*

*while (temp != nullptr)*

*{*

*std::cout << temp->data << " ";*

*temp = temp->right;*

*}*

*std::cout << std::endl;*

*}*

*};*

*int main()*

*{*

*doublee l{};*

*l.insertAtHead(2);*

*l.insertAtHead(1);*

*l.insertAtEnd(4);*

*l.insertAtEnd(5);*

*l.insertAfter(4, 8);*

*l.display();*

*l.reverse();*

*l.display();*

*return 0;*

*}*

**DOUBLEE QUEUE**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* right{nullptr};*

*node\* left{nullptr};*

*node\* tail{nullptr};*

*};*

*class queue*

*{*

*private:*

*node\* head{nullptr};*

*public:*

*bool empty() {return head == nullptr;}*

*node\* search(int key)*

*{*

*node\* temp{head};*

*while(temp != nullptr)*

*{*

*if(temp->data == key)*

*return temp;*

*temp = temp->right;*

*}*

*return nullptr;*

*}*

*void insertAtEnd(int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*head = tail = newer;*

*return;*

*}*

*tail->right = newer; //newer right is nullptr by default*

*newer->left = tail;*

*tail = newer;*

*}*

*void delAtHead()*

*{*

*if(empty())*

*{*

*std::cout << "empty!\n";*

*return;*

*}*

*node\* temp{head->right};*

*delete head;*

*if(temp != nullptr) //if multi nodes,*

*temp->left = nullptr; //update left of next node(new head)*

*else*

*tail = nullptr;*

*head = temp; //update head(incase of single node temp would be null)*

*}*

*void display()*

*{*

*node\* temp = head;*

*while (temp != nullptr)*

*{*

*std::cout << temp->data << " ";*

*temp = temp->right;*

*}*

*std::cout << std::endl;*

*}*

*};*

*int main()*

*{*

*queue l{};*

*l.insertAtEnd(4);*

*l.insertAtEnd(7);*

*l.insertAtEnd(98);*

*l.display();*

*l.delAtHead();*

*l.delAtHead();*

*l.insertAtEnd(3);*

*l.display();*

*l.display();*

*return 0;*

*}*

**CIRCULAR**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* next{nullptr};*

*};*

*class circList*

*{*

*private:*

*node\* head{nullptr}; //using head is hella inefficient, increases time complexity so we use last*

*public:*

*bool empty() {return head == nullptr;}*

*node\* search(int val)*

*{*

*node\* temp{head};*

*do*

*{*

*if(temp->data == val)*

*return temp;*

*temp = temp->next;*

*}*

*while(temp != head);*

*return nullptr;*

*}*

*void insertAtHead(int val) //sir ver better*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*newer->next = newer; //point to itself if empty*

*else*

*{*

*node\* temp{head};*

*while(temp->next != head) //while temp isnt node before head(last)*

*temp = temp->next;*

*temp->next = newer;*

*newer->next = head;*

*}*

*head = newer;*

*}*

*void insertAtEnd(int val) //sir ver better*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*newer->next = newer; //insertAtHead();*

*head = newer;*

*}*

*else*

*{*

*node\* temp{head};*

*while(temp->next != head)*

*temp = temp->next;*

*temp->next = newer;*

*newer->next = head;*

*} //only difference from atHead is exclusion of head = newer;*

*}*

*void insertAfter(int val, int key)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*newer->next = newer; //insertAtHead();*

*head = newer;*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*delete newer;*

*return;*

*}*

*/\* if(found->next == head) //no need,*

*insertAtEnd(val); \*/*

*newer->next = found->next;*

*found->next = newer;*

*}*

*void insertBefore(int val, int key)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*newer->next = newer; //insertAtHead();*

*head = newer;*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found\n";*

*delete newer;*

*return;*

*}*

*if(found == head)*

*{*

*insertAtHead(val);*

*return;*

*}*

*node\* temp{head};*

*while(temp->next != found)*

*temp=temp->next;*

*temp->next = newer;*

*newer->next = found;*

*}*

*void deleteAtHead()*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* temp = head;*

*if(head->next == head) //for single node*

*{*

*std::cout << "deleted value: " << temp->data << '\n';*

*delete temp;*

*head = nullptr;*

*return;*

*}*

*node\* temp2{head};*

*while(temp2->next != head)*

*temp2=temp2->next;*

*head = temp->next;*

*temp2->next = head;*

*std::cout << "deleted value: " << temp->data << '\n';*

*delete temp;*

*}*

*void deleteAtEnd()*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* temp1{head};*

*if(head->next == head)*

*{*

*std::cout << "deleted value: " << temp1->data << '\n';*

*delete temp1;*

*head = nullptr;*

*return;*

*}*

*node\* temp2{nullptr};*

*while(temp1->next != head)*

*{*

*temp2 = temp1;*

*temp1 = temp1->next;*

*}*

*temp2->next = head;*

*std::cout << "deleted value: " << temp1->data << '\n';*

*delete temp1;*

*}*

*void deleteAfter(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*else if(found->next == head)*

*{*

*deleteAtHead();*

*return;*

*}*

*node\* temp{found->next};*

*found->next = temp->next;*

*delete temp;*

*}*

*void deleteBefore(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head)*

*{*

*deleteAtEnd();*

*return;*

*}*

*if(found == head->next)*

*{*

*deleteAtHead();*

*return;*

*}*

*node\* temp{head};*

*while(temp->next->next != found)*

*temp = temp->next;*

*node\* temp2 = temp->next;*

*temp->next = found;*

*delete temp2;*

*}*

*void deleteParticular(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == NULL)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head)*

*deleteAtHead();*

*else if(found->next == head)*

*deleteAtEnd();*

*else*

*{*

*node\* temp{head};*

*while(temp->next != found)*

*temp = temp->next;*

*temp->next = found->next;*

*delete found;*

*}*

*}*

*/\* void deleteAll()*

*{*

*while(head->next != head)*

*{*

*node\* temp{head->next};*

*deleteParticular(temp->data);*

*}*

*deleteAtHead();*

*} \*/*

*void deleteAll()*

*{*

*node\* temp{head};*

*do*

*{*

*node\* temp2{temp};*

*temp = temp->next;*

*delete temp2;*

*}*

*while(temp != head);*

*head = nullptr;*

*}*

*void reverse()*

*{*

*node\* prev{head};*

*node\* curr{head};*

*node\* next{head->next};*

*while(prev->next != head)*

*prev = prev->next;*

*do*

*{*

*curr->next = prev;*

*prev = curr;*

*curr = next;*

*next = next->next;*

*}*

*while(curr != head);*

*head->next = prev;*

*head = prev;*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node\* temp{head};*

*do*

*{*

*std::cout << temp->data << " ";*

*temp = temp->next;*

*}*

*while(temp != head);*

*std::cout << '\n';*

*}*

*};*

*int main()*

*{*

*circList l{};*

*l.insertAtHead(2);*

*l.insertAtHead(3);*

*l.insertAtEnd(1);*

*l.insertAfter(11, 1);*

*l.insertBefore(7, 11);*

*l.show();*

*l.reverse();*

*l.show();*

*return 0;*

*}*

**QUEUE/STACK**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* next{nullptr};*

*};*

*class stack*

*{*

*private:*

*node\* last{nullptr}; //since it is circular, we consider the first node as last, and the next one as first(which is head normally)*

*public: //so insert at head here is insert at last(first node), insertAtEnd at last node which loops back to last again*

*bool empty() {return last == nullptr;}*

*void insert(int val) //at head*

*{*

*node\* newer{new node};*

*newer->data = val;*

*//1 4 3 2 stack insert(at head)(can be read as 4 3 2 1 where 4 can be taken as head and 1 is last)*

*if(empty())*

*last = newer; //1 2 3 4 queue insert(at end)*

*else*

*newer->next = last->next;*

*last->next = newer;*

*// last = newer; //add this for queue(at end)*

*}*

*void pop() //at head*

*{*

*if(empty())*

*return;*

*node\* temp{last->next};*

*if(last == temp)*

*last = nullptr;*

*else*

*last->next = temp->next;*

*std::cout << "del val: " << temp->data << '\n';*

*delete temp;*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node\* temp{last->next};*

*do*

*{*

*std::cout << temp->data << " ";*

*temp = temp->next;*

*}*

*while(temp != last->next);*

*std::cout << '\n';*

*}*

*};*

*int main()*

*{*

*stack s{};*

*s.insert(1);*

*s.insert(2);*

*s.insert(3);*

*s.insert(4);*

*s.show();*

*s.pop();*

*s.show();*

*}*

**JOSEPHUS**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* next{nullptr};*

*};*

*class cirList*

*{*

*private:*

*node\* head{nullptr};*

*public:*

*bool empty() {return head==nullptr;}*

*node\* search(int val)*

*{*

*node\* temp{head};*

*do*

*{*

*if(temp->data == val)*

*return temp;*

*temp = temp->next;*

*}*

*while(temp != head);*

*return nullptr;*

*}*

*void insertAtEnd(int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if(empty())*

*{*

*newer->next = newer;*

*head = newer;*

*}*

*else*

*{*

*node\* temp{head};*

*while(temp->next != head)*

*temp = temp->next;*

*temp->next = newer;*

*newer->next = head;*

*}*

*}*

*void delParticular(int key)*

*{*

*if(empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if(found == nullptr)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if(found == head && singleNode())*

*{*

*std::cout << head->data << " deleted\n";*

*delete head;*

*head = nullptr;*

*return;*

*}*

*if(found == head)*

*{*

*head = found->next;*

*}*

*node\* temp{head};*

*while(temp->next != found)*

*temp = temp->next;*

*temp->next = found->next;*

*std::cout << found->data << " deleted\n";*

*delete found;*

*}*

*void show()*

*{*

*if(empty())*

*return;*

*node\* temp{head};*

*do*

*{*

*std::cout << temp->data << " ";*

*temp = temp->next;*

*}*

*while(temp != head);*

*std::cout << '\n';*

*}*

*bool singleNode() {return head->next == head;}*

*void josephusWin(int k)*

*{*

*node\* temp{head}; //last-> next*

*while(!singleNode())*

*{*

*for(int i{0}; i < k-1; i++)*

*{*

*temp = temp->next;*

*}*

*node\* todel{temp};*

*temp = temp->next;*

*delParticular(todel->data);*

*}*

*}*

*};*

*void makeList(cirList& l, int n)*

*{*

*for(int i{1}; i <= n; i++)*

*l.insertAtEnd(i);*

*}*

*int main()*

*{*

*std::cout << "enter no of players: ";*

*int n{};*

*std::cin >> n;*

*cirList l{};*

*makeList(l, n);*

*std::cout << "enter no of players to skip before a kill: ";*

*int k{};*

*std::cin >> k;*

*l.show();*

*l.josephusWin(k);*

*l.show();*

*return 0;*

*}*

**JOSEPHUS NON MEMBER FUNC**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* next{nullptr};*

*};*

*class cirList*

*{*

*private:*

*node\* head{nullptr};*

*public:*

*bool empty() { return head == nullptr; }*

*node\* search(int val)*

*{*

*node\* temp{head};*

*do*

*{*

*if (temp->data == val)*

*return temp;*

*temp = temp->next;*

*}*

*while (temp != head);*

*return nullptr;*

*}*

*void insertAtEnd(int val)*

*{*

*node\* newer{new node};*

*newer->data = val;*

*if (empty())*

*{*

*newer->next = newer;*

*head = newer;*

*}*

*else*

*{*

*node\* temp{head};*

*while (temp->next != head)*

*temp = temp->next;*

*temp->next = newer;*

*newer->next = head;*

*}*

*}*

*void delParticular(int key)*

*{*

*if (empty())*

*{*

*std::cout << "empty!" << '\n';*

*return;*

*}*

*node\* found{search(key)};*

*if (found == nullptr)*

*{*

*std::cout << key << " not found." << '\n';*

*return;*

*}*

*if (found == head && singleNode())*

*{*

*std::cout << head->data << " deleted\n";*

*delete head;*

*head = nullptr;*

*return;*

*}*

*if (found == head)*

*{*

*head = found->next;*

*}*

*node\* temp{head};*

*while (temp->next != found)*

*temp = temp->next;*

*temp->next = found->next;*

*std::cout << found->data << " deleted\n";*

*delete found;*

*}*

*void show()*

*{*

*if (empty())*

*return;*

*node\* temp{head};*

*do*

*{*

*std::cout << temp->data << " ";*

*temp = temp->next;*

*}*

*while (temp != head);*

*std::cout << '\n';*

*}*

*bool singleNode() { return head->next == head; }*

*};*

*void makeList(cirList& l, int n)*

*{*

*for (int i{1}; i <= n; i++)*

*l.insertAtEnd(i);*

*}*

*// Non-member function to solve Josephus problem*

*void josephusWin(cirList& l, int n, int k)*

*{*

*int remaining = n; // To keep track of the number of remaining players*

*node\* current = l.search(1); // Start from the first player (node with data 1)*

*while (remaining > 1) // Continue until only one player remains*

*{*

*// Find the k-th player to be eliminated*

*for (int i = 1; i < k; i++) // Move k-1 steps*

*{*

*current = current->next;*

*}*

*// Delete the k-th player*

*node\* toDelete = current;*

*current = current->next; // Move to the next player*

*l.delParticular(toDelete->data); // Delete the node*

*remaining--; // Decrease the remaining players count*

*}*

*}*

*int main()*

*{*

*std::cout << "Enter number of players: ";*

*int n{};*

*std::cin >> n;*

*cirList l{};*

*makeList(l, n);*

*std::cout << "Enter number of players to skip before a kill: ";*

*int k{};*

*std::cin >> k;*

*l.show();*

*josephusWin(l, n, k);*

*l.show();*

*return 0;*

*}*

**BINARY SEARCH TREE**

*#include <iostream>*

*#include <stack>*

*struct node*

*{*

*int data{};*

*node \*right{nullptr}, \*left{nullptr};*

*node(int val) : data{val}, right{nullptr}, left{nullptr} {};*

*};*

*class bst*

*{*

*private:*

*node\* root{nullptr};*

*public:*

*bool empty() {return root == nullptr;}*

*node\* getRoot() {return root;}*

*bool isLeft(int val)*

*{*

*node\* par{getParent(val)};*

*if(par == nullptr)*

*return false;*

*return (par->left != nullptr && par->left->data == val);*

*}*

*bool isRight(int val)*

*{*

*node\* par{getParent(val)};*

*if(par == nullptr)*

*return false;*

*return (par->right != nullptr && par->right->data == val); //if right is null right->data will give exception if null isnt checked*

*}*

*node\* getParent(int val)*

*{*

*node\* cur{root}, \*par{nullptr};*

*while(cur != nullptr)*

*{*

*if(val < cur->data)*

*{*

*par = cur;*

*cur = cur->left;*

*}*

*else if(val > cur->data)*

*{*

*par = cur;*

*cur = cur->right;*

*}*

*else*

*return par;*

*}*

*std::cout << val << " not found\n";*

*return nullptr;*

*}*

*node\* search(int val)*

*{*

*node\* temp{root};*

*while(temp != nullptr)*

*{*

*if(val < temp->data)*

*temp = temp->left;*

*else if(val > temp->data)*

*temp = temp->right;*

*else*

*return temp;*

*}*

*return nullptr;*

*}*

*void insert(int val)*

*{*

*if(empty())*

*{*

*root = new node(val);*

*return;*

*}*

*node\* cur{root}, \*par{nullptr};*

*while(cur != nullptr)*

*{*

*par = cur;*

*if(val < cur->data)*

*cur = cur->left;*

*else if(val > cur->data)*

*cur = cur->right;*

*else*

*{*

*std::cout << "duplicate val\n";*

*return;*

*}*

*}*

*if(val < par->data)*

*par->left = new node(val);*

*else*

*par->right = new node(val);*

*}*

*void del(int val)*

*{*

*node\* cur{search(val)};*

*node\* par{getParent(val)};*

*if(cur->left == nullptr && cur->right == nullptr) //if leaf*

*{*

*if(isLeft(val))*

*par->left = nullptr;*

*else*

*par->right = nullptr;*

*delete cur;*

*}*

*else if(cur->left == nullptr) //if one child*

*{*

*node\* child{cur->right};*

*if(isLeft(val))*

*par->left = child;*

*else*

*par->right = child;*

*delete cur;*

*return;*

*}*

*else if(cur->right == nullptr) //if one child*

*{*

*node\* child{cur->left};*

*if(isLeft(val))*

*par->left = child;*

*else*

*par->right = child;*

*delete cur;*

*return;*

*}*

*else //if two child*

*{*

*node\* temp{cur->right};*

*while(temp->left != nullptr)*

*temp = temp->left;*

*int dTemp = temp->data;*

*del(temp->data);*

*cur->data = dTemp;*

*}*

*}*

*void preOrder(node\* temp)*

*{*

*if(temp == nullptr)*

*return;*

*std::cout << temp->data << ' ';*

*preOrder(temp->left);*

*preOrder(temp->right);*

*}*

*void preOrderStack()*

*{*

*if(root == nullptr) //empty*

*return;*

*std::stack<node\*> s{};*

*s.push(root);*

*while(!s.empty())*

*{*

*node\* temp{s.top()}; //we take the top at each iteration as temp*

*s.pop(); //*

*std::cout << temp->data << ' ';*

*if(temp->right)*

*s.push(temp->right); //push right first*

*if(temp->left) //then left*

*s.push(temp->left); //so top will have left*

*}*

*}*

*void inOrder(node\* temp)*

*{*

*if(temp == nullptr)*

*return;*

*inOrder(temp->left);*

*std::cout << temp->data << ' ';*

*inOrder(temp->right);*

*}*

*void inOrder() { inOrder(root); } //PUBLIC WRAPPER FUNCTION*

*void inOrderStack()*

*{*

*if(root == nullptr)*

*return;*

*std::stack<node\*> s{};*

*node\* temp{root};*

*while(!s.empty() || temp != nullptr) //till s is empty && temp == nullptr*

*{*

*while(temp != nullptr)*

*{*

*s.push(temp);*

*temp = temp->left;*

*}*

*temp = s.top();*

*s.pop();*

*std::cout << temp->data << ' ';*

*temp = temp->right;*

*}*

*}*

*void postOrder(node\* temp)*

*{*

*if(temp == nullptr)*

*return;*

*postOrder(temp->left);*

*postOrder(temp->right);*

*std::cout << temp->data << ' ';*

*}*

*void postOrderStack()*

*{*

*if (root == nullptr)*

*return;*

*std::stack<node\*> s1, s2;*

*s1.push(root);*

*// Create a reversed post-order traversal using two stacks*

*while (!s1.empty())*

*{*

*node\* temp = s1.top();*

*s1.pop();*

*s2.push(temp);*

*if (temp->left) s1.push(temp->left);*

*if (temp->right) s1.push(temp->right);*

*}*

*// Print nodes in correct post-order*

*while (!s2.empty())*

*{*

*std::cout << s2.top()->data << " ";*

*s2.pop();*

*}*

*}*

*int count(node\* temp)*

*{*

*if(temp == nullptr)*

*return 0;*

*return 1 + count(temp->left) + count(temp->right);*

*}*

*int sum(node\* temp)*

*{*

*if(temp == nullptr)*

*return 0;*

*return temp->data + sum(temp->left) + sum(temp->right);*

*}*

*};*

*int main()*

*{*

*bst t{};*

*t.insert(5);*

*t.insert(2);*

*t.insert(7);*

*t.insert(1);*

*t.insert(10);*

*t.insert(3);*

*t.insert(11);*

*t.inorder();*

*std::cout << std::endl;*

*t.del(5);*

*t.preOrder(t.getRoot());*

*return 0;*

*}*

**RIGHT THREADED TREE**

*#include <iostream>*

*struct node*

*{*

*int data{};*

*node\* left{nullptr};*

*node\* right{nullptr};*

*bool rthread{true};*

*node(int val) : data{val}, next{nullptr}, right{nullptr}, rthread{true} {}*

*};*

*class bst*

*{*

*private:*

*node\* root{nullptr};*

*public:*

*bool empty() {return root == nullptr;}*

*node\* search(int val)*

*{*

*node\* temp{root};*

*while(temp != nullptr)*

*{*

*if(val < temp->data)*

*temp = temp->left;*

*else if(val > temp->data)*

*{*

*if(!temp->rthread)*

*temp = temp->right;*

*else*

*break;*

*}*

*else*

*return temp;*

*}*

*return nullptr;*

*}*

*void insert(int val)*

*{*

*if(empty())*

*root = new node(val);*

*node\* cur{root}, \* par{nullptr};*

*while(cur != nullptr)*

*{*

*par = cur;*

*if(val < cur->data)*

*cur = cur->left;*

*else if(val > cur->data)*

*{*

*if(!cur->rthread)*

*cur = cur->right;*

*else*

*break;*

*}*

*else*

*return;*

*}*

*node\* newer{new node(val)};*

*if(val < par)*

*{*

*par->left = newer;*

*newer->right = par;*

*}*

*else*

*{*

*newer->right = par->right;*

*par->rthread = false;*

*par->right = newer;*

*}*

*}*

*void traversal()*

*{*

*node\* temp{root};*

*while(temp->left != nullptr)*

*temp = temp->left;*

*while(temp != nullptr)*

*{*

*std::cout << temp->data << " ";*

*if(temp->rthread)*

*temp = temp->right;*

*else*

*{*

*temp = temp->right;*

*while(temp->left != nullptr)*

*temp = temp->left;*

*}*

*}*

*}*

*};*

**END**