**North South University - Spring 2023**

Course: CSE225L Assessment: HW 2

Section: 06 NSU ID: 2211424642 Name: Joy Kumar Ghosh

// Task-1 Codes

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  const int MAX\_ITEMS = 5;  template <class ItemType>  class UnsortedType{  public:  UnsortedType();  void MakeEmpty();  bool IsFull();  int LengthIs();  void InsertItem(ItemType);  void DeleteItem(ItemType);  void RetrieveItem(ItemType &, bool &);  void ResetList();  void GetNextItem(ItemType &);  private:  int length;  ItemType info[MAX\_ITEMS];  int currentPos;  }; |
| template <class ItemType>  UnsortedType<ItemType>::UnsortedType(){  length = 0;  currentPos = -1;  }  template <class ItemType>  void UnsortedType<ItemType>::MakeEmpty(){  length = 0;  }  template <class ItemType>  bool UnsortedType<ItemType>::IsFull(){  return (length == MAX\_ITEMS);  }  template <class ItemType>  int UnsortedType<ItemType>::LengthIs(){  return length;  }  template <class ItemType>  void UnsortedType<ItemType>::ResetList(){  currentPos = -1;  }  template <class ItemType>  void UnsortedType<ItemType>::GetNextItem(ItemType &item){  currentPos++;  item = info[currentPos];  }  template <class ItemType>  void UnsortedType<ItemType>::RetrieveItem(ItemType &item, bool &found){  int location = 0;  bool moreToSearch = (location < length);  found = false;  while (moreToSearch && !found){  if(item == info[location]){  found = true;  item = info[location];  }  else{  location++;  moreToSearch = (location < length);  }  }  }  template <class ItemType>  void UnsortedType<ItemType>::InsertItem(ItemType item){  info[length] = item;  length++;  }  template <class ItemType>  void UnsortedType<ItemType>::DeleteItem(ItemType item){  int location = 0;  while (item != info[location])  location++;  info[location] = info[length - 1];  length--;  } |
| class studentInfo{  private:  int id;  string name;  double cgpa;  public:  studentInfo();  studentInfo(int, string, double);  bool operator==(const studentInfo& other) const;  bool operator!=(const studentInfo& other) const;  void print();  }; |
| studentInfo::studentInfo(){  }  studentInfo::studentInfo(int i, string n, double c){  id = i;  name = n;  cgpa = c;  }  bool studentInfo::operator==(const studentInfo& other) const{  return id == other.id;  }  bool studentInfo::operator!=(const studentInfo& other) const{  return id != other.id;  }  void studentInfo::print(){  cout << id << ", " << name << ", " << cgpa << endl;  } |
| int main()  {  //supporting variable  int i, id;  double cgpa;  string name;  studentInfo item;  bool isFound = false;  //list of object  UnsortedType<studentInfo> studentList;  //taking 5 input  cout << "Enter 5 Students Records(ID <> Name <> CGPA):\n";  for(i = 0; i < 5; i++){  cin >> id;  cin >> name;  cin >> cgpa;  item = studentInfo(id, name, cgpa);  studentList.InsertItem(item);  }  //deleting records  cout << "\nEnter ID to delete: ";  cin >> id;  item = studentInfo(id, "searching", 0.0);  studentList.DeleteItem(item);  //retrieving records  cout << "\n\nEnter ID for retrieving records: ";  cin >> id;  item = studentInfo(id, "searching", 0.0);  studentList.RetrieveItem(item, isFound);  if(isFound){  cout << "Item is Found\n";  item.print();  }  else{  cout << "Item is not Found\n";  }  //printing the list  cout << "\nAll Students Records:\n";  for(i = 0, studentList.ResetList(); i < studentList.LengthIs(); i++){  studentList.GetNextItem(item);  item.print();  }  cout << endl;  return 0;  } |
|  |

// Task-2 Codes

|  |
| --- |
| #include <iostream>  using namespace std;  const int MAX\_ITEMS = 5;  template <class ItemType>  class SortedType{  public :  SortedType();  void MakeEmpty();  bool IsFull();  int LengthIs();  void InsertItem(ItemType);  void DeleteItem(ItemType);  void RetrieveItem(ItemType &, bool &);  void ResetList();  void GetNextItem(ItemType &);  private:  int length;  ItemType info[MAX\_ITEMS];  int currentPos;  }; |
| template <class ItemType>  SortedType<ItemType>::SortedType() {  length = 0;  currentPos = - 1;  }  template <class ItemType>  void SortedType<ItemType>::MakeEmpty() {  length = 0;  }  template <class ItemType>  bool SortedType<ItemType>::IsFull() {  return (length == MAX\_ITEMS);  }  template <class ItemType>  int SortedType<ItemType>::LengthIs() {  return length;  }  template <class ItemType>  void SortedType<ItemType>::ResetList() {  currentPos = - 1;  }  template <class ItemType>  void  SortedType<ItemType>::GetNextItem(ItemType &item){  currentPos++;  item = info [currentPos];  }  template <class ItemType>  void SortedType<ItemType>::InsertItem(ItemType item){  int location = 0;  bool moreToSearch = (location < length);  while (moreToSearch) {  if(item > info[location]) {  location++;  moreToSearch = (location < length);  } else if(item < info[location])  moreToSearch = false;  }  for (int index = length; index > location;  index--)  info[index] = info[index - 1];  info[location] = item;  length++;  }  template <class ItemType>  void SortedType<ItemType>::DeleteItem(ItemType item){  int location = 0;  while (item != info[location])  location++;  for (int index = location + 1; index < length;  index++)  info[index - 1] = info[index];  length--;  }  template <class ItemType>  void SortedType<ItemType>::RetrieveItem(ItemType &item, bool &found){  int midPoint, first = 0, last = length - 1;  bool moreToSearch = (first <= last);  found = false;  while (moreToSearch && !found) {  midPoint = (first + last) / 2;  if(item < info[midPoint]) {  last = midPoint - 1;  moreToSearch = (first <= last);  } else if(item > info[midPoint]) {  first = midPoint + 1;  moreToSearch = (first <= last);  } else {  found = true;  item = info[midPoint];  }  }  } |
| class timeStamp{  private:  int ss, mm, hh;  public:  timeStamp();  timeStamp(int, int, int);  bool operator==(const timeStamp& other) const;  bool operator!=(const timeStamp& other) const;  bool operator>(const timeStamp& other) const;  bool operator<(const timeStamp& other) const;  void print();  }; |
| timeStamp::timeStamp(){  }  timeStamp::timeStamp(int s, int m, int h){  ss = s;  mm = m;  hh = h;  }  bool timeStamp::operator==(const timeStamp& other) const{  return (ss == other.ss && mm == other.mm && hh == other.hh);  }  bool timeStamp::operator!=(const timeStamp& other) const{  return !(ss == other.ss && mm == other.mm && hh == other.hh);  }  bool timeStamp::operator>(const timeStamp& other) const{  if(hh > other.hh) return true;  else if (hh == other.hh){  if(mm > other.mm) return true;  else if(mm == other.mm){  if(ss > other.ss) return true;  }  }  return false;  }  bool timeStamp::operator<(const timeStamp& other) const{  if(hh < other.hh) return true;  else if (hh == other.hh){  if(mm < other.mm) return true;  else if(mm == other.mm){  if(ss < other.ss) return true;  }  }  return false;  }  void timeStamp::print(){  if(ss > 9) cout << ss << ":";  else cout << "0" << ss << ":";  if(mm > 9) cout << mm << ":";  else cout << "0" << mm << ":";  if(hh > 9) cout << hh << endl;  else cout << "0" << hh << endl;  } |
| int main()  {  //supporting variable  int i, ss, mm, hh;  timeStamp item;  //list of object  SortedType<timeStamp> timeList;  //taking 5 input  cout << "Enter 5 Time Records(ss <> mm <> hh):\n";  for(i = 0; i < 5; i++){  cin >> ss;  cin >> mm;  cin >> hh;  item = timeStamp(ss, mm, hh);  timeList.InsertItem(item);  }  //deleting records  cout << "\nEnter Time to delete(ss <> mm <> hh): ";  cin >> ss;  cin >> mm;  cin >> hh;  item = timeStamp(ss, mm, hh);  timeList.DeleteItem(item);  //printing the list  cout << "\nAll Time Records:\n";  for(i = 0, timeList.ResetList(); i < timeList.LengthIs(); i++){  timeList.GetNextItem(item);  item.print();  }  cout << endl;  return 0;  } |
|  |

// Task-3 Codes

|  |
| --- |
| #include <iostream>  using namespace std;  //class declaration  template <class ItemType>  class UnsortedType  {  struct NodeType  {  ItemType info;  NodeType\* next;  };  public:  UnsortedType();  ~UnsortedType();  bool IsFull();  int LengthIs();  void MakeEmpty();  void RetrieveItem(ItemType &, bool &);  void InsertItem(ItemType);  void DeleteItem(ItemType);  void ResetList();  void GetNextItem(ItemType&);  private:  NodeType\* listData;  int length;  NodeType\* currentPos;  }; |
| //implementation  template <class ItemType>  UnsortedType<ItemType>::UnsortedType(){  length = 0;  listData = NULL;  currentPos = NULL;  }  template <class ItemType>  int UnsortedType<ItemType>::LengthIs(){  return length;  }  template<class ItemType>  bool UnsortedType<ItemType>::IsFull(){  NodeType\* location;  try  {  location = new NodeType;  delete location;  return false;  }  catch(bad\_alloc& exception)  {  return true;  }  }  template <class ItemType>  void UnsortedType<ItemType>::InsertItem(ItemType item){  NodeType\* location;  location = new NodeType;  location->info = item;  location->next = listData;  listData = location;  length++;  }  template <class ItemType>  void UnsortedType<ItemType>::DeleteItem(ItemType item){  NodeType\* location = listData;  NodeType\* tempLocation;  if (item == listData->info){  tempLocation = location;  listData = listData->next;  }  else{  while (!(item==(location->next)->info))  location = location->next;  tempLocation = location->next;  location->next = (location->next)->next;  }  delete tempLocation;  length--;  }  template <class ItemType>  void UnsortedType<ItemType>::RetrieveItem(ItemType &item, bool &found){  NodeType\* location = listData;  bool moreToSearch = (location != NULL);  found = false;  while (moreToSearch && !found){  if (item == location->info)  found = true;  else{  location = location->next;  moreToSearch = (location != NULL);  }  }  }  template <class ItemType>  void UnsortedType<ItemType>::MakeEmpty(){  NodeType\* tempPtr;  while (listData != NULL){  tempPtr = listData;  listData = listData->next;  delete tempPtr;  }  length = 0;  }  template <class ItemType>  UnsortedType<ItemType>::~UnsortedType(){  MakeEmpty();  }  template <class ItemType>  void UnsortedType<ItemType>::ResetList(){  currentPos = NULL;  }  template <class ItemType>  void UnsortedType<ItemType>::GetNextItem(ItemType &item){  if (currentPos == NULL)  currentPos = listData;  else  currentPos = currentPos->next;  item = currentPos->info;  } |
| int main()  {  //list  UnsortedType<int> list01, list02, list03;  //supporting variable  int m, n, i, j, item, item01, item02;  //taking input of list 01  cout << "Enter size of list 01: ";  cin >> m;  cout << "Enter " << m << " item with space:\n";  for(i = 0; i < m; i++){  cin >> item;  list01.InsertItem(item);  }  //taking input of list 02  cout << "\nEnter size of list 02: ";  cin >> n;  cout << "Enter " << n << " item with space:\n";  for(i = 0; i < n; i++){  cin >> item;  list02.InsertItem(item);  }  //combining two list in list 03  list01.ResetList();  list02.ResetList();  i = j = 0;  list01.GetNextItem(item01);  list02.GetNextItem(item02);  while(1){  if(item01 > item02){  list03.InsertItem(item01);  i++;  if(i < list01.LengthIs())  list01.GetNextItem(item01);  else break;  }  else{  list03.InsertItem(item02);  j++;  if(j < list02.LengthIs())  list02.GetNextItem(item02);  else break;  }  }  while(i < list01.LengthIs()){  list03.InsertItem(item01);  i++;  if(i < list01.LengthIs())  list01.GetNextItem(item01);  }  while(j < list02.LengthIs()){  list03.InsertItem(item02);  j++;  if(j < list02.LengthIs())  list02.GetNextItem(item02);  }  //printing list  cout << "\nCombined List:\n";  for(i = 0, list03.ResetList(); i < list03.LengthIs(); i++){  list03.GetNextItem(item);  cout << item << " ";  }  cout << endl;  return 0;  } |
|  |

// Task-4 Codes

|  |
| --- |
| #include <iostream>  using namespace std;  const int MAX\_ITEMS = 5;  class FullStack{  // Exception class thrown  // by Push when stack is full.  };  class EmptyStack{  // Exception class thrown  // by Pop and Top when stack is empty.  };  template <class ItemType>  class StackType{  public:  StackType();  bool IsFull();  bool IsEmpty();  void Push(ItemType);  void Pop();  ItemType Top();  private:  int top;  ItemType items[MAX\_ITEMS];  }; |
| template <class ItemType>  StackType<ItemType>::StackType(){  top = -1;  }  template <class ItemType>  bool StackType<ItemType>::IsEmpty(){  return (top == -1);  }  template <class ItemType>  bool StackType<ItemType>::IsFull(){  return (top == MAX\_ITEMS-1);  }  template <class ItemType>  void StackType<ItemType>::Push(ItemType newItem){  if(IsFull())  throw FullStack();  top++;  items[top] = newItem;  }  template <class ItemType>  void StackType<ItemType>::Pop(){  if(IsEmpty())  throw EmptyStack();  top--;  }  template <class ItemType>  ItemType StackType<ItemType>::Top(){  if (IsEmpty())  throw EmptyStack();  return items[top];  } |
| // Function to check if a set of parentheses is balanced or not  bool isBalanced(string expression){  StackType<char> temp;  for(int i = 0; i < expression.length(); i++){  if(expression[i] == '('){  temp.Push(expression[i]);  }  else if(expression[i] == ')'){  if(temp.IsEmpty() || temp.Top() != '('){  return false;  }  temp.Pop();  }  }  return temp.IsEmpty();  } |
| int main()  {  string expression;  for(int i = 0; i < 4; i++){  cout << "Enter Expression: ";  cin >> expression;  if(isBalanced(expression)){  cout << "Results: Balanced." << endl << endl;  }  else{  cout << "Results: Not Balanced." << endl << endl;  }  }  return 0;  } |
|  |

// Task-5 Codes

|  |
| --- |
| #include <iostream>  using namespace std;  class FullQueue{  };  class EmptyQueue{  };  template<class ItemType>  class QueType{  public:  QueType();  QueType(int max);  ~QueType();  void MakeEmpty();  bool IsEmpty();  bool IsFull();  void Enqueue(ItemType);  void Dequeue(ItemType&);  private:  int front;  int rear;  ItemType\* items;  int maxQue;  }; |
| template<class ItemType>  QueType<ItemType>::QueType(int max){  maxQue = max + 1;  front = maxQue - 1;  rear = maxQue - 1;  items = new ItemType[maxQue];  }  template<class ItemType>  QueType<ItemType>::QueType(){  maxQue = 501;  front = maxQue - 1;  rear = maxQue - 1;  items = new ItemType[maxQue];  }  template<class ItemType>  QueType<ItemType>::~QueType(){  delete [] items;  }  template<class ItemType>  void QueType<ItemType>::MakeEmpty(){  front = maxQue - 1;  rear = maxQue - 1;  }  template<class ItemType>  bool QueType<ItemType>::IsEmpty(){  return (rear == front);  }  template<class ItemType>  bool QueType<ItemType>::IsFull(){  return ((rear+1)%maxQue == front);  }  template<class ItemType>  void QueType<ItemType>::Enqueue(ItemType newItem){  if(IsFull())  throw FullQueue();  else{  rear = (rear +1) % maxQue;  items[rear] = newItem;  }  }  template<class ItemType>  void QueType<ItemType>::Dequeue(ItemType& item){  if(IsEmpty())  throw EmptyQueue();  else{  front = (front + 1) % maxQue;  item = items[front];  }  } |
| int main()  {  int number;  string binary;  QueType<string> binaryQueue;  cout << "Enter a number: ";  cin >> number;  binaryQueue.Enqueue("1");  for(int i = 0; i < number; i++){  binaryQueue.Dequeue(binary);  cout << binary << endl;  binaryQueue.Enqueue(binary + "0");  binaryQueue.Enqueue(binary + "1");  }  return 0;  } |
|  |

// Task-6 Codes

|  |
| --- |
| #include <iostream>  #include <string>  #include <sstream>  using namespace std;  class FullStack{  };  class EmptyStack{  };  template <class ItemType>  class StackType{  struct NodeType{  ItemType info;  NodeType\* next;  };  public:  StackType();  ~StackType();  void Push(ItemType);  void Pop();  ItemType Top();  bool IsEmpty();  bool IsFull();  private:  NodeType\* topPtr;  }; |
| template <class ItemType>  StackType<ItemType>::StackType(){  topPtr = NULL;  }  template <class ItemType>  bool StackType<ItemType>::IsEmpty(){  return (topPtr == NULL);  }  template <class ItemType>  ItemType StackType<ItemType>::Top(){  if(IsEmpty())  throw EmptyStack();  else  return topPtr->info;  }  template <class ItemType>  bool StackType<ItemType>::IsFull(){  NodeType\* location;  try{  location = new NodeType;  delete location;  return false;  }  catch(bad\_alloc& exception){  return true;  }  }  template <class ItemType>  void StackType<ItemType>::Push(ItemType newItem){  if(IsFull())  throw FullStack();  else{  NodeType\* location;  location = new NodeType;  location->info = newItem;  location->next = topPtr;  topPtr = location;  }  }  template <class ItemType>  void StackType<ItemType>::Pop(){  if(IsEmpty())  throw EmptyStack();  else{  NodeType\* tempPtr;  tempPtr = topPtr;  topPtr = topPtr->next;  delete tempPtr;  }  }  template <class ItemType>  StackType<ItemType>::~StackType(){  NodeType\* tempPtr;  while(topPtr != NULL) {  tempPtr = topPtr;  topPtr = topPtr->next;  delete tempPtr;  }  } |
| bool isValidExpression(string expression)  {  StackType<char> temp;  for(int i = 0; i < expression.length(); i++){  if(expression[i] == ' ') continue;  if(expression[i] == '+' || expression[i] == '-' ||expression[i] == '\*' ||expression[i] == '/'){  int j = i+1;  while(expression[j] == ' ') j++;  if(expression[j] == '+' || expression[j] == '-' || expression[j] == '\*' || expression[j] == '/')  return false;  }  if(expression[i] == '('){  temp.Push(expression[i]);  }  else if(expression[i] == ')'){  if(temp.IsEmpty() || temp.Top() != '('){  return false;  }  temp.Pop();  }  }  return temp.IsEmpty();  } |
| int getPrecedence(string ch)  {  if(ch == "/") return 3;  else if(ch == "\*") return 2;  else if(ch == "+" || ch == "-") return 1;  else return -1;  } |
| bool isOperator(string op)  {  if(op == "+" || op == "-" || op == "\*" || op == "/" || op == "(" || op == ")") return true;  else return false;  } |
| string infixToPostfix(string infix)  {  StackType<string> stackOfOperator;  string postfix = "";  stringstream splitInfix(infix);  string inf;  while(splitInfix >> inf){  if(!isOperator(inf)){  postfix += (inf + " ");  }  else if(inf == "(") stackOfOperator.Push("(");  else if(inf == ")"){  while(stackOfOperator.Top() != "("){  postfix += (stackOfOperator.Top() + " ");  stackOfOperator.Pop();  }  stackOfOperator.Pop();  }  else{  while(!stackOfOperator.IsEmpty() && getPrecedence(inf) <= getPrecedence(stackOfOperator.Top())){  postfix += (stackOfOperator.Top() + " ");  stackOfOperator.Pop();  }  stackOfOperator.Push(inf);  }  }  while(!stackOfOperator.IsEmpty()){  postfix += (stackOfOperator.Top() + " ");  stackOfOperator.Pop();  }  return postfix;  } |
| double evaluate(double a, double b, char ch)  {  switch(ch)  {  case '+':  return a + b;  case '-':  return a - b;  case '\*':  return a \* b;  case '/':  return a / b;  default:  return -1;  }  } |
| double evaluatePostfix(string postfix)  {  double a, b;  StackType<double> stackOfOperand;  stringstream splitInfix(postfix);  string inf;  while(splitInfix >> inf){  if(!isOperator(inf)){  stackOfOperand.Push(stod(inf));  }  else{  b = stackOfOperand.Top();  stackOfOperand.Pop();  a = stackOfOperand.Top();  stackOfOperand.Pop();  stackOfOperand.Push(evaluate(a, b, inf[0]));  }  }  return stackOfOperand.Top();  } |
| int main()  {  string expression;  for(int i = 0; i < 4; i++){  cout << "Enter expression: ";  getline(cin, expression);  if(isValidExpression(expression)){  cout << "Result: ";  cout << evaluatePostfix(infixToPostfix(expression)) << "\n\n";  }  else{  cout << "Invalid Expression!!\n\n";  }  }  return 0;  } |
|  |

// Task-7 Codes

|  |
| --- |
| #include <iostream>  using namespace std;  class FullQueue  {  };  class EmptyQueue  {  };  template <class ItemType>  class QueType{  struct NodeType{  ItemType info;  NodeType\* next;  };  public:  QueType();  ~QueType();  void MakeEmpty();  void Enqueue(ItemType);  void Dequeue(ItemType&);  bool IsEmpty();  bool IsFull();  private:  NodeType \*front, \*rear;  }; |
| template <class ItemType>  QueType<ItemType>::QueType(){  front = NULL;  rear = NULL;  }  template <class ItemType>  bool QueType<ItemType>::IsEmpty(){  return (front == NULL);  }  template<class ItemType>  bool QueType<ItemType>::IsFull(){  NodeType\* location;  try{  location = new NodeType;  delete location;  return false;  }  catch(bad\_alloc& exception){  return true;  }  }  template <class ItemType>  void QueType<ItemType>::Enqueue(ItemType newItem){  if(IsFull())  throw FullQueue();  else{  NodeType\* newNode;  newNode = new NodeType;  newNode->info = newItem;  newNode->next = NULL;  if (rear == NULL)  front = newNode;  else  rear->next = newNode;  rear = newNode;  }  }  template <class ItemType>  void QueType<ItemType>::Dequeue(ItemType& item){  if(IsEmpty())  throw EmptyQueue();  else{  NodeType\* tempPtr;  tempPtr = front;  item = front->info;  front = front->next;  if (front == NULL)  rear = NULL;  delete tempPtr;  }  }  template <class ItemType>  void QueType<ItemType>::MakeEmpty(){  NodeType\* tempPtr;  while(front != NULL){  tempPtr = front;  front = front->next;  delete tempPtr;  }  rear = NULL;  }  template <class ItemType>  QueType<ItemType>::~QueType(){  MakeEmpty();  } |
| void QueueCopy(QueType<int>& where, QueType<int>& from)  {  QueType<int> tempQueue;  int item;  while(!from.IsEmpty()){  from.Dequeue(item);  tempQueue.Enqueue(item);  }  while(!tempQueue.IsEmpty()){  tempQueue.Dequeue(item);  where.Enqueue(item);  from.Enqueue(item);  }  } |
| int minCoins(QueType<int>& coins, int amount)  {  // Base case  if (amount == 0) {  return 0;  }  // Initialize the minimum number of coins to the maximum value  int minimumCoins = 99999;  QueType<int> tempQueue;  QueueCopy(tempQueue, coins);  // Try every coin that has smaller value than amount  while (!tempQueue.IsEmpty()) {  int coin;  tempQueue.Dequeue(coin);  if (coin <= amount) {  // Find the minimum number of coins required to make the remaining amount  int subCoins = minCoins(coins, amount - coin);  // Update the minimum number of coins  minimumCoins = min(minimumCoins, subCoins + 1);  }  }  return minimumCoins;  } |
| int main()  {  // Get the number of coin types  cout << "Enter test case number: ";  int testCase;  cin >> testCase;  for(int i = 0; i < testCase; i++){  cout << "Enter number of Coins: ";  int numberOfCoins;  cin >> numberOfCoins;  // Get the coin values  QueType<int> coins;  cout << "Enter " << numberOfCoins << " coins with space: ";  for (int i = 0; i < numberOfCoins; i++) {  int coin;  cin >> coin;  coins.Enqueue(coin);  }  // Get the amount of money  cout << "Enter Amount: ";  int amount;  cin >> amount;  // Find the minimum number of coins required  int minimumCoins = minCoins(coins, amount);  // Print the minimum number of coins  cout << "Minimum Coins required to make " << amount << " is: " << minimumCoins << endl << endl << endl;  }  return 0;  } |
|  |
| SS Difference: Upgraded to windows 11 |
| Just stuck on a Loop. |
| Explanation: This is called stack overflow. In this algorithm 200 amount is too large. For that recursive call gets a stack overflow situation. To solve this problem, we need to use more efficient algorithm. After some more research on google I found something that can solve this problem. We can fix it using a dynamic array called vector. |
| Here is the modified code for minCoins function |
| int minCoins(QueType<int>& coins, int amount) {  std::vector<int> dp(amount + 1, INT\_MAX); // Initialize a vector to store minimum coin counts for amounts from 0 to 'amount'  dp[0] = 0; // Base case: 0 coins needed to make an amount of 0  for (int i = 1; i <= amount; i++) {  QueType<int> tempQueue;  QueueCopy(tempQueue, coins);  while (!tempQueue.IsEmpty()) {  int coin;  tempQueue.Dequeue(coin);  if (coin <= i && dp[i - coin] != INT\_MAX) {  dp[i] = std::min(dp[i], dp[i - coin] + 1);  }  }  }  if (dp[amount] == INT\_MAX) {  return -1; // Return -1 to indicate that the amount cannot be made with the available coins  }  return dp[amount];  } |
|  |
| Now its optimized. Most efficient code. |