|  |  |
| --- | --- |
| **Data Structures & Algorithms**  Diploma in CSF, IT  Year 2 (2023/24) Semester 3 | **Week 3** |
| **1-2 Hours** |
| **Tutorial 3 – Linked List** | |

1. Explain the purpose of the following operators

1. **&**

|  |
| --- |
| Used to get address of a variable, returning the memory address of a variable  Example:  int num = 42;  int\* ptr = &num; // `&num` gives the memory address of `num`  cout << "Address of num: " << &num << endl; // Outputs the address of `num`  cout << "Value of ptr: " << ptr << endl; // `ptr` stores the address of `num` |

1. **\***

|  |
| --- |
| \* used to dereference a pointer, it accesses the value stored at the memory address the pointer points to  This allows user to access the object or value that the pointer points to  Example: int num = 42;  int\* ptr = &num; // `ptr` points to `num`  cout << "Value of num: " << \*ptr << endl; // `\*ptr` accesses the value stored at `ptr` |

1. **new**

|  |
| --- |
| Dynamically allocate memory on the heap |

2. Assuming the Node structure is declared as:

struct Node

{

string item; // to store the data item

Node \*next; // pointer to point to next node

};

1. Draw diagrams to show what happens in computer memory when the following statements are executed.

Node node1;

Node \*node2;

|  |
| --- |
| Node node1; |
| Node \*node2; |

1. Write the statements to:

(i) store “Kevin” in node1.

(ii) initialize the next pointer in node1 to NULL

|  |
| --- |
| // (i)store “Kevin” in node1.  node1->item = “Kevin”;  // (ii)initialize the next pointer in node1 to NULL  node1->next = nullptr; |

1. Write the statements to:
2. create a Node object and set node2 to point to it
3. store “Vivian” in the Node object pointed to by node2.

(iii) initialize the next pointer in Node object, pointed to by node2, to NULL

|  |
| --- |
| //(i) create a Node object and set node2 to point to it  Node\* node3 = new Node;  node2->next = node3  //(ii) store “Vivian” in the Node object pointed to by node2.  node3->item = "Vivian";  //(iii)  initialize the next pointer in Node object, pointed to by node2, to NULL  node3->next = NULL; |

1. What happens when the following statements are executed?

Node node3 = node1;

|  |
| --- |
| node3 will be a new node object, separate from node1  Item value of node1 is copied into node 3  Next pointer value of node 1 will be copied to node 3, that is nullptr |

Node \*node4 = node2;

|  |
| --- |
| node4 declared as a pointer to a Node, assigned value of node2.  node2 is also a pointer to a Node object  Both node4 and node2 will point to the same Node object in memory (right now is still nullptr) |

3. The specification of a List ADT implemented using Pointers is given below.

|  |
| --- |
| *// List.h - - Specification of List ADT (implemented using Pointers)*  #pragma once  #include<string>  #include<iostream>  using namespace std;  typedef string ItemType;  Q  class List  {  private:  struct Node  {  ItemType item; // item  Node \*next; // pointer pointing to next item  };  Node \*firstNode; // point to the first item  int size; // number of items in the list  public:  // constructor  List();  *// add an item to the back of the list (append)*  bool add(ItemType item);  *// add an item at a specified position in the list (insert)*  bool add(int index, ItemType item);  *// remove an item at a specified position in the list*  void remove(int index);  *// get an item at a specified position of the list (retrieve)*  ItemType get(int index);  *// check if the list is empty*  bool isEmpty();  *// check the size of the list*  int getLength();  }; |

Implement the following operations of the List ADT

(a) bool add(ItemType item)

|  |
| --- |
| // Add an item to the back of the list (append)  bool List::add(ItemType item)  {      Node \*newNode = new Node;      newNode->item = item;      newNode->next = nullptr;      if (isEmpty())          firstNode = newNode; // Insert new node      else      {          Node \*temp = firstNode; // Traverse to the last node          while (temp->next != nullptr)              temp = temp->next; // Move to the next node          temp->next = newNode;  // Link to the new node      }      size++;      return true;  } |

(b) bool add(int index, ItemType item);

|  |
| --- |
| // Add an item at the specified position in the list (insert)  bool List::add(int index, ItemType item)  {      if (index < 1 || index > size + 1) // Index out of range          return false;      Node \*newNode = new Node;      newNode->item = item;      if (index == 1)      {          newNode->next = firstNode;          firstNode = newNode;      }      else      {          Node \*current = firstNode;          // Iterate through each node until we reach the node before the index          for (int i = 1; i < index - 1; i++)          {              current = current->next; // Move to the next node          }          newNode->next = current->next; // Link the new node to the next node          current->next = newNode;       // Link current node to new node      }      size++;      return true;  } |

(c) int getLength()

|  |
| --- |
| // Check the size of the list  int List::getLength()  {      return size;  } |