Question 1 (30 marks)

The specification of a **pointer-based List ADT** is given in Figure 1:

|  |
| --- |
| #include<iostream>  using namespace std;  typedef char ItemType;  class List  {  private:  struct Node  {  ItemType item; // item  Node\* next; // pointer pointing to next node  };  Node\* front; // pointer pointing to front node  public:  //constructor  **List();**    // add a new item at a specified position in the list  // 0 <= index < size  **bool add(int index, ItemType item);**    // remove an item at a specified position in the list  // 0 <= index < size  **void remove(int index);**    // get an item at a specified position in the list  // 0 <= index < size  **ItemType get(int index);**  // returns the number of occurrences of item in the list  **int count(ItemType item);**  // recursive function that  // returns the number of occurrences of item in the list  **int countR(ItemType item);**  // reverses the list  **void reverse();**    // recursive function that reverses the list  **void reverseR();**  // prints all the items in the list  **void display();**  }; |

Figure 1: Specification of a pointer-based List ADT

(a) Write the iterative **count(ItemType item)** function that returns the number of occurrences of the item in the list.

int List::count(ItemType item) {

int count = 0;

if (!isEmpty()) {

Node\* tempNode = firstNode;

while (tempNode != NULL) { // want to move until we reach the empty node (the end)

if (item == tempNode->item) {

count++;

}

tempNode = tempNode->next;

}

return count;

}

else {

return 0;

}

}

This is correct.

(b) Write the iterative **reverse()** function that reverses the items in the list.

**void List::reverse()**

**{**

**Node\* previous, \* current, \* next;**

**previous = NULL;**

**current = firstNode;**

**next = NULL;**

**if (isEmpty())**

**{**

**cout << "empty";**

**}**

**while (current != NULL)**

**{**

**next = current->next; //The next pointer will be stored with the address of the next node after current.**

**current->next = previous; //The current node's next will store the address of previous.**

**previous = current; //Previous pointer will now point to current address.**

**current = next; //Current pointer will now point to next address.**

**}**

**firstNode = previous;**

**}**

(c) Write the recursive **countR(ItemType item)** function that returns the number of occurrences of the item in the list.

int List::countR(ItemType item)

{

countR2(firstNode, item); // need the first node, so create a helper function

}

// for recursion, the problem needs to get smaller until the base case.

//If the parameters is not enough you need to create the parameters.

int List::countR2(Node\* temp, ItemType item)

{

if (temp == NULL) // base case

return 0;

else

{

if (temp->item == item)

return 1 + countR2(temp->next, item); //temp->next, makes the list smaller.

else

return countR2(temp->next, item);

}

}

Note: Use helper function when parameters arent enough.

(d) Write the recursive **reverseR()** function that reverses the items in the list.

void List::reverseR()

{

reverseR2(front);

}

List::Node\* List::reverseR2(Node\* node)

{

if (node == NULL) // list is empty

return NULL;

if (node->next == NULL) { // if there is only one node

front = node;

return node;

}

List::Node\* node1 = reverseR2(node->next); // reverse 2nd node onwards

node1->next = node; // adjust last node of reversed list to point it backwards

node->next = NULL; // node is now last node, so its next is NULL

return node;

}

Question 2 (30 marks)

The specification of an ***array-based* List ADT** is given in Figure 2:

|  |
| --- |
| const int MAX\_SIZE = 100;  typedef int ItemType;  class **List**  {  private:  ItemType itemArray[MAXSIZE]; // items in the list  int size; // number of items in the list    public:  //constructor  **List();**    // add an item to the back of the list (append)  // pre : none  // post: item is added to the back of the list  // size is increased by 1  **bool add(ItemType item);**    // add an item at a specified position in the list  // pre : 1 <= index <= size  // post: item is added to the specified position in the list  // size is increased by 1  **bool add(int index, ItemType item);**  // add an item at a specified position in the list using recursion  **bool addR(int index, ItemType item);**    // remove an item at a specified position in the list  // pre: 1 <= index <= size  // post: item is removed from specified position in list  // size is decreased by 1  **void remove(int index);**    // get an item at a specified position of the list  // pre : 1 <= index <= size  // post: none  **ItemType get(int index);**    // count and return the number of items in the list  // pre : none  // post: none  **int getLength();**  }; |

Figure 2: Specification of an array-based List ADT

1. Implement the **iterative** **add(int index, ItemType item)** function for the List ADT.

(10 marks)

bool add(int index, ItemType item)

{

bool success = (index>=1) && (index <=size+1) && (size < MAX\_SIZE);

if (success)

{

for (int pos = size; pos >= index; pos--) //shifting

itemArray[pos] = itemArray[pos-1];

itemArray[index-1] item; // store the item

size++; // increase size of list

}

return success;

}

1. Implement the **recursive** **addR(int index, ItemType item)** function for the List ADT.

bool List::addR(int index, ItemType item) {

if (index < size) {

addR(index + 1, items[index]);

items[index] = item;

return true;

}

else if (index == size) {

items[index] = item;

size++;

return true;

}

}

(10 marks)

bool add(int index, ItemType item)

{

bool success = (index>=1) && (index <=size+1) && (size < MAX\_SIZE);

if (success)

{

// for (int pos = size; pos >= index; pos--) //shifting needs recursion

// itemArray[pos] = itemArray[pos-1];

addR2(index, size, item);

itemArray[index-1] = item; // store the item

size++; // increase size of list

}

return success;

}

void addR2(int index, int pos, ItemType item)

{

if (index == pos) // base case

return;

else

{

itemArray[pos] = itemArray[pos-1];

addR2(index, pos-1, item);

}

}

(c) What is the time-complexity of the **add()** function in (a). Justify your answer.

(4 marks)

The time complexity of the add() function in (a) is O(n).

This is because the function needs to shift all the items from the index to the end of the list by 1 position to make room for the new item.

In the worst case where the index is 0, the number of items to be shifted is n (where n is the number of items in the list)

(d) Two lists, listA and listB, have been created using the List ADT. Assuming that listA and listB are empty at the beginning, state the contents of listA and listB after each of statements in the table below are executed.

(6 marks)

|  |  |
| --- | --- |
| Statements | Contents in ListA and ListB |
| listA.add(6); | listA :\_\_\_\_\_\_6\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| listA.add(8); | listA :\_\_6,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| listA.add(2, 5); | listA :\_\_6,5,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_\_\_\_\_\_\_\_\_\_\_\_\_   |  |  | | --- | --- | |  |  |   \_\_\_\_\_\_\_\_\_\_\_\_ |
| listB.add(listA.get(3)); | listA :\_\_\_\_6,5,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_\_8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| listB.add(listA.get(1)); | listA :\_\_\_\_6,5,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_8,6\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| listB.add(listA.get(2)); | listA :\_\_\_\_6,5,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_8,6,5\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| listA.remove(2); | listA :\_\_\_\_6,8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  listB :\_\_8,6,5\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

If you are asked to draw the list.

Use table.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 6 |  |  |  |  |  |  |  |

|  |
| --- |
|  |

Question 3 (20 marks)

(a) A **CircularQueue** is an abstract data type (ADT) which is similar to a regular queue data structure, but the front of the queue is “joined” to the back of the queue to form a “circle”.

The specification of an ***array-based*** CircularQueue ADT is given below. It stores the indexes of the front and back of the queue. When an element is enqueued, the back index is incremented; when an element is dequeued, the front index is incremented. The indexes “wrap around” back to zero (0) when the end of the array is reached.

|  |
| --- |
| const int MAX\_SIZE = 50;  typedef int ItemType;  class CircularQueue  {  private:  ItemType items[MAX\_SIZE];  int front; // index of the front of the queue  int back; // index of the back of the queue  public:  // constructor  **CircularQueue();**  // insert item  // returns false if queue is full, true otherwise  **bool enqueue(ItemType item);**  // retrieve and remove item  // returns false of queue is empty, true otherwise  **bool dequeue(ItemType &item);**  // check if the queue is empty  **bool isEmpty();**  // check if the queue is full  **bool isFull();**  // other functions not shown  }; |

Figure 2b: Specification of an array-based CircularQueue ADT

(a) Implement the **isFull()** function such that it has time complexity of O(1).

**bool CircularQueue::isFull()**

**{**

**if ((rear+1)%MAX\_SIZE == front)**

**{**

**return true;**

**}**

**else {**

**return false;**

**}**

}

(5 marks)

(b) Implement the **enqueue()** function such that it has time complexity of O(1). It returns false if the CircularQueue is full, and true otherwise.

Yo go sleep bro.

**bool CircularQueue::enqueue(ItemType item)**

**{**

**//queue is full**

**if ((rear + 1) % MAX\_SIZE == front)**

**{**

**cout << "Queue is full \n";**

**return false;**

**}**

**else**

**{**

**//first element inserted**

**if (front == -1)**

**{**

**front = 0;**

**}**

**//insert element at rear**

**rear = (rear + 1) % MAX\_SIZE;**

**items[rear] = item;**

**return true;**

**}**

**}**

(8 marks)

(c) Implement the **dequeue()** function such that it has time complexity of O(1). It returns false if the CircularQueue is empty, and true otherwise.

**bool CircularQueue::dequeue(ItemType& item)**

**{**

**if (isEmpty() )**

**{**

**cout << "Queue is empty\n";**

**return false;**

**}**

**else {**

**//only one element**

**if (front == rear)**

**{**

**front = rear = -1;**

**}**

**else**

**{**

**front = (front + 1) % MAX\_SIZE;**

**}**

**return true;**

**}**

**}**

(7 marks)

(d) Show the queue as the following code executes.

You are to show clearly the queue at the end of each queue operation.

int x = 6;

int y = 3;

queue.enqueue(x);

queue.enqueue(y);

x = queue.getFront();

queue.dequeue();

queue.enqueue(x + y);

queue.enqueue(16);

queue.dequeue();

queue.enqueue(y - 5);

The original queue of size 10 with front = 5 and back = 9 is given below.

(10 marks)

index

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  | 10 | 4 | 6 | 3 | 5 |

Front = 5, back = 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 |  |  |  |  | 10 | 4 | 6 | 3 | 5 |

front = 5, back = 0

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 |  |  |  | 10 | 4 | 6 | 3 | 5 |

front = 5, back = 1

X = 10

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 |  |  |  | 10 | 4 | 6 | 3 | 5 |

front = 6, back = 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 | 9 |  |  | 10 | 4 | 6 | 3 | 5 |

front = 6, back = 2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 | 9 | 16 |  | 10 | 4 | 6 | 3 | 5 |

front = 6, back = 3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 | 9 | 16 |  | 10 | 4 | 6 | 3 | 5 |

front = 7, back = 3

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 3 | 9 | 16 | -2 | 10 | 4 | 6 | 3 | 5 |

front = 7, back = 4

Check if full is: front == (back+1)%size in case the back+1 becomes > size of queue

Front = 5, back =4 check 5==5 true. Full.

QUESTION 4 (20 marks)

(a) A Random Access Queue has the following operations:

* 1. pushRight(x): to add a new item, x, to the tail
  2. popLeft(): to remove the item at the head
  3. elementAt(i): to retrieve the item at position i without removing it. i = 0 gives the item at the head, i = 1 the following element and so on. If there are less than i elements, the value -999 is returned.

1. Suppose this data structure is implemented using a **linked list**.

State the time complexity of each of the three operations. Explain your answer.

(4 marks)

(A) = O(n) if there is no tail pointer, need to traverse the length of the queue to get to the last item.

= O(1) if there is a tail pointer, then just adjust the tail pointer.

(B) = O(1) since need to only adjust the pointer to the head.

(C) = O(n) since need to traverse to the ith position to retrieve the item at position i.

1. Suppose this data structure is implemented using an **array**. Figure 2(b)(ii) shows the queue holding 4 integer items, stored within an array of size 8.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  |  | 63 | 59 | 22 | 94 |  |
|  |  |  |  | head |  |  | tail |  |

Figure 2(b)(ii): Sample data in array implementation

Write an implementation of the elementAt() function which has a time complexity of O(1).

The function prototype of elementAt() is

int elementAt(int i);

You may assume that head and tail are attributes of the data structure.

(4 marks)

elementAt(3) return 94

int elementAt(int i)

{

if (i < size)

return array[(head+i)%8];

else

return –999; // not found

}

(b) (i) Write a recursive function, **displayReverseOrder()**, to display an array of names in the reverse order. Indicate the base case and recursive step clearly.

(8 marks)

// display an array of names in the reverse order

// n is the size of the array

**void displayReverseOrder(string array[], int n)**

**{**

**// base case**

**if (n==1)**

**cout << array[n-1];**

**// recursive step**

**else**

**{**

**cout << array[n-1] << endl;**

**displayReverseOrder(array, n-1);**

**}**

**}**

Time complexity of iterative and recursive displayReverseOrder() is the same O(n)

(ii) Discuss one advantage and one disadvantage of implementing the **displayReverseOrder()** using recursion instead of using iteration.

(4 marks)

Advantage:

More natural way of thinking. Don’t have to set up a variable to do the looping.

Disadvantage:

Uses more memory because each function call requires a new stack activation record.