|  |
| --- |
| **Data Structures and Algorithms**  ***Section*: BSCE2021 Assignment # 10 & 11 *Total marks*: 100**  ***Name*** : ***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Roll number* : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |

***Submission:***

• *Email instructor or TA if there are any questions. You cannot look at others’ solutions or use others’ solutions, however, you can discuss it with each other. Plagiarism will be dealt with according to the course policy.*

*• Submission after due time will not be accepted.*

**There should be a Report explaining your code and highlighting the results. Follow this naming convention for your report RollNumber\_Assignment#.pdf e.g BSCE21000\_Assignment7.pdf.**

**TASK: A**

implement the Quick Sort by using a linked list and also make a menu-driven program. You should also include the insertion and deletion functionality of the linked list.

**Do not swap data. swap nodes**

Program should be menu driven. the program should continue to run until the exit option is selected. Handle all corner cases.

|  |
| --- |
| **FUNCTION.H:**  #include <iostream>  using namespace std;  class Node { //made a class of node public:  int data; //declared data in public  Node \*nextPtr;   Node(int val) {  data = val; //made a constructor to set values  nextPtr = NULL;  }   void setNextPtr(Node \*n) {  nextPtr = n; //set the next ptr  }   Node \*getNextPtr() {  return nextPtr; //get the value of next ptr  }   ~Node() {  nextPtr = nullptr;  } };  class linkList { //making 1 other class of linklist public:  Node \*tail;  Node \*head; //made some pointers  int count;   linkList() {  count = 0;  head = nullptr; //declared them to zero  tail = nullptr;  }   ~linkList() {  head = nullptr;  tail = nullptr;  }   Node \*getHead() {  return head;  }   void append(int value) {  Node \*temp = new Node(value); //declaring  temp->data = value;  temp->nextPtr = nullptr; //initializing the next ptr in the next of the new initialized node  if (head == nullptr) {  head = temp; //if the head is null then store the temp in head  } else {  Node \*temp1 = head; //else make a new node  while (temp1->nextPtr != nullptr) //iterate it till the node is not null  temp1 = temp1->nextPtr; //store the temp to next ptr address  temp1->nextPtr = temp; //store pointer to the last one  }  count++;  }   void prepend(int value) {  Node \*temp = new Node(value); //make a new node  temp->data = value; //store the value  temp->nextPtr = head; //point the pointer to head  head = temp; //making the head that new temp  count++;  }   void insertAtIndex(int index, int value) {  Node \*temp = new Node(value); //making a new node  temp->data = value; //storing value  temp->nextPtr = nullptr; //pointing to null  if (index <= 0) {  cout << "INDEX MUST BE POSITIVE."  << endl; //checking that index is not zero or negative  }  if (index == 1) {  temp->nextPtr = head; //is index is 1 then store the head in next ptr and make the temp that head  temp = head;  return;  } else {  Node \*newTemp = head; //making a new node  int i = 1;  while (i < index - 1) { //iterating to the place before the index  newTemp = newTemp->nextPtr;  i++;  }  if (newTemp != nullptr) { //after that checking that node is not null  temp->nextPtr = newTemp->nextPtr; //storing the new temp to the previous temp  newTemp->nextPtr = temp; //making the new temp that previous temp  } else {  cout << "THE PREVIOUS NODE IS NOT NULL." << endl;  }  }  count++;  }   void deleteAtEnd() {  if (head == nullptr) {  cout << "ARRAY IS EMPTY." << endl; //checking is the head is null  }  if (head != nullptr) { //if head is not nul  if (head->nextPtr == nullptr) { //if the next to head is null  head = nullptr; //make the head null  } else {  Node \*temp = head; //making a new node  while (temp->nextPtr->nextPtr != nullptr) //if the next to next address is not null  temp = temp->nextPtr; //then store the next address to temp  Node \*temp1 = temp->nextPtr; //store the temp address to new node  temp->nextPtr = nullptr; //make it null and delete the new node  delete temp1;  }  }  count--;  }   void deleteAtStart() {  if (head == nullptr) { //is head is null the array is empty  cout << "THE ARRAY IS EMPTY." << endl;  } else {  Node \*newTemp = head; //making a new node and giving it address of head  head = head->nextPtr; //giving head the address the head is pointing to  delete newTemp; // and deleting the temp  }  count--;  }   static Node \*getTail(Node \*cur) {  while (cur != nullptr && cur->nextPtr != nullptr)  cur = cur->nextPtr;  return cur;  }   void deleteAtIndex(int index) {  if (index <= 0) {  cout << "THE INDEX MUST BE POSITIVE." << endl; //checking the index  }  if (index == 1 && head != nullptr) { //if head is 1 or not null then make a new node  Node \*temp = head;  head = head->nextPtr; //and store the address of next ptr to that new node  delete temp; //delete new node  } else {  Node \*temp1 = head; //make a new node give it the address of head  int i = 1;  while (i < index - 1) { //iterate till 1 less position of the index  temp1 = temp1->nextPtr; //and storing the addresses  i++;  }  Node \*toDelete = temp1->nextPtr; //making new node and storing the address of the ptr of next ptr  temp1->nextPtr = temp1->nextPtr->nextPtr; //storing the value of next to next value to delete the middle one  delete toDelete; //deleting the ptr  }  count--;  }   void display() const {  Node \*temp = head; //declaring  while (temp != nullptr) {  cout << temp->data << "\t"; //displaying the data  temp = temp->nextPtr; //storing the next address  cout << " "; //displaying space  }  }   static Node \*partition(Node \*low, Node \*high) {  Node \*pivotPointer = low;  Node \*currentPoint = low; //initializing when the first and the last node with low   int pivotPoint = high->data;  while (low != high) { //if low is not equal to high  if (low->data < pivotPoint) { //if low of data is less than the high of data stored in pivot point  pivotPointer = currentPoint; //storing  int value = currentPoint->data; //storing the data  currentPoint->data = low->data; //then swapping  low->data = value;  currentPoint = currentPoint->nextPtr;  }  low = low->nextPtr; //storing the next node   }  int value = currentPoint->data; //swapping  currentPoint->data = pivotPoint;  high->data = value;  return pivotPointer; //returning pivot pointer to use in quick sort function   }   void quicksort(Node \*low, Node \*high) {  if (low == high) { //if low and high are same then return   return;  }  Node \*temp = partition(low, high); //to use the returning pointer we make a new node   quicksort(low, temp); //then pass that node to sort half   quicksort(temp->nextPtr, high); //then the other  }  };  **MAIN.CPP:**  #include <iostream> #include "Functions.h"  using namespace std;  int main() {  Node n(5); //making object  n.setNextPtr(0);  n.getNextPtr(); //calling  linkList l; //making object  l.display();  int opt1; //declaring  do {   cout << "\nCHOOSE OPTIONS." << endl;  cout << "1.APPEND." << endl;  cout << "2.PREPEND." << endl;  cout << "3.ADD AT SPECIFIC INDEX." << endl; //displaying options  cout << "4.DELETE AN ELEMENT FROM THE START OF THE LIST." << endl;  cout << "5.DELETE AN ELEMENT FROM THE LAST OF THE LIST." << endl;  cout << "6.DELETE AN ELEMENT FROM THE SPECIFIC INDEX OF THE LIST." << endl;  cout << "7.QUICK SORT."<<endl;  cout << "8.EXIT." << endl;  cin >> opt1;  if (opt1 == 1) {  l.append(3);  l.append(4);  l.append(5);  l.append(6);  l.display();  }   if (opt1 == 2) {  l.prepend(4);  l.prepend(0);  l.prepend(1);  l.prepend(2);  l.display();  }  if (opt1 == 3) {  int index;  int value;  cout << "ENTER THE INDEX AT WHICH YOU WANT TO ADD THE VALUE." << endl;  cin >> index;  cout << "ENTER THE NUMBER YOU WANT TO ADD IN LIST = " << endl;  cin >> value; //taking value  l.insertAtIndex(index, value); //calling  l.display();  }  if (opt1 == 4) {  l.deleteAtStart(); //calling  l.display();  }  if (opt1 == 5) {  l.deleteAtEnd(); //calling  l.display();  }  if (opt1 == 6) {  int index;  cout << "ENTER THE INDEX FROM WHICH YOU WANT TO DELETE THE VALUE." << endl;  cin >> index;  l.deleteAtIndex(index);  l.display(); //calling  }  if (opt1 == 7) {  Node \*temp=l.getHead();  l.quicksort(temp,l.getTail(temp));  l.display();  }  if (opt1 == 8) {  cout << "YOU CHOOSE TO EXIT." << endl; //displaying  exit(3);  }  } while (opt1 >= 1 && opt1 <= 7); //condition to terminate the do while loop   return 0; } |

**TASK: B**

implement the Radix Sort by using a linked list and also make a menu-driven program. You should also include the insertion and deletion functionality of the linked list.

**Do not swap data. swap nodes**

Program should be menu driven. the program should continue to run until the exit option is selected. Handle all corner cases.

|  |
| --- |
| #include <iostream>  using namespace std;  class Node { //made a class of node public:  int data; //declared data in public  Node \*nextPtr;   Node(int val) {  data = val; //made a constructor to set values  nextPtr = NULL;  }   void setNextPtr(Node \*n) {  nextPtr = n; //set the next ptr  }   Node \*getNextPtr() {  return nextPtr; //get the value of next ptr  }   ~Node() {  nextPtr = nullptr;  } };  class linkList { //making 1 other class of linklist public:  Node \*tail;  Node \*head; //made some pointers  int count;   linkList() {  count = 0;  head = nullptr; //declared them to zero  tail = nullptr;  }   ~linkList() {  head = nullptr;  tail = nullptr;  }   Node \*getHead() const {  return head;  }   void append(int value) {  Node \*temp = new Node(value); //declaring  temp->data = value;  temp->nextPtr = nullptr; //initializing the next ptr in the next of the new initialized node  if (head == nullptr) {  head = temp; //if the head is null then store the temp in head  } else {  Node \*temp1 = head; //else make a new node  while (temp1->nextPtr != nullptr) //iterate it till the node is not null  temp1 = temp1->nextPtr; //store the temp to next ptr address  temp1->nextPtr = temp; //store pointer to the last one  }  count++;  }   void prepend(int value) {  Node \*temp = new Node(value); //make a new node  temp->data = value; //store the value  temp->nextPtr = head; //point the pointer to head  head = temp; //making the head that new temp  count++;  }   void insertAtIndex(int index, int value) {  Node \*temp = new Node(value); //making a new node  temp->data = value; //storing value  temp->nextPtr = nullptr; //pointing to null  if (index <= 0) {  cout << "INDEX MUST BE POSITIVE."  << endl; //checking that index is not zero or negative  }  if (index == 1) {  temp->nextPtr = head; //is index is 1 then store the head in next ptr and make the temp that head  temp = head;  return;  } else {  Node \*newTemp = head; //making a new node  int i = 1;  while (i < index - 1) { //iterating to the place before the index  newTemp = newTemp->nextPtr;  i++;  }  if (newTemp != nullptr) { //after that checking that node is not null  temp->nextPtr = newTemp->nextPtr; //storing the new temp to the previous temp  newTemp->nextPtr = temp; //making the new temp that previous temp  } else {  cout << "THE PREVIOUS NODE IS NOT NULL." << endl;  }  }  count++;  }   void deleteAtEnd() {  if (head == nullptr) {  cout << "ARRAY IS EMPTY." << endl; //checking is the head is null  }  if (head != nullptr) { //if head is not nul  if (head->nextPtr == nullptr) { //if the next to head is null  head = nullptr; //make the head null  } else {  Node \*temp = head; //making a new node  while (temp->nextPtr->nextPtr != nullptr) //if the next to next address is not null  temp = temp->nextPtr; //then store the next address to temp  Node \*temp1 = temp->nextPtr; //store the temp address to new node  temp->nextPtr = nullptr; //make it null and delete the new node  delete temp1;  }  }  count--;  }   int deleteAtStart() {  int value;  if (head == nullptr) { //is head is null the array is empty  cout << "THE ARRAY IS EMPTY." << endl;  } else {  Node \*newTemp = head; //making a new node and giving it address of head  value= newTemp->data;  head = head->nextPtr; //giving head the address the head is pointing to  delete newTemp; // and deleting the temp  }  count--;  return value;  }   static Node \*getTail(Node \*cur) {  while (cur != nullptr && cur->nextPtr != nullptr)  cur = cur->nextPtr;  return cur;  }   void deleteAtIndex(int index) {  if (index <= 0) {  cout << "THE INDEX MUST BE POSITIVE." << endl; //checking the index  }  if (index == 1 && head != nullptr) { //if head is 1 or not null then make a new node  Node \*temp = head;  head = head->nextPtr; //and store the address of next ptr to that new node  delete temp; //delete new node  } else {  Node \*temp1 = head; //make a new node give it the address of head  int i = 1;  while (i < index - 1) { //iterate till 1 less position of the index  temp1 = temp1->nextPtr; //and storing the addresses  i++;  }  Node \*toDelete = temp1->nextPtr; //making new node and storing the address of the ptr of next ptr  temp1->nextPtr = temp1->nextPtr->nextPtr; //storing the value of next to next value to delete the middle one  delete toDelete; //deleting the ptr  }  count--;  }   void display() const {  Node \*temp = head; //declaring  while (temp != nullptr) {  cout << temp->data << "\t"; //displaying the data  temp = temp->nextPtr; //storing the next address  cout << " "; //displaying space  }  }   static Node \*partition(Node \*low, Node \*high) {  Node \*pivotPointer = low;  Node \*currentPoint = low; //initializing when the first and the last node with low  int pivotPoint = high->data;  while (low != high) { //if low is not equal to high  if (low->data <  pivotPoint) { //if low of data is less than the high of data stored in pivot point  pivotPointer = currentPoint; //storing  int value = currentPoint->data; //storing the data  currentPoint->data = low->data; //then swapping  low->data = value;  currentPoint = currentPoint->nextPtr;  }  low = low->nextPtr; //storing the next node  }  int value = currentPoint->data; //swapping  currentPoint->data = pivotPoint;  high->data = value;  return pivotPointer; //returning pivot pointer to use in quick sort function  }   void quicksort(Node \*low, Node \*high) {  if (low == high) { //if low and high are same then return  return;  }  Node \*temp = partition(low, high); //to use the returning pointer we make a new node  quicksort(low, temp); //then pass that node to sort half  quicksort(temp->nextPtr, high); //then the other  }   int isEmpty() const {  if (head == nullptr) { //if head is null then return 1 else false  return 1;  } else {  return 0;  }  }  static int max(linkList &l)  {  int value; //declaring  int digit=0; //initializing  Node \*temp = l.getHead() ;  value = temp->data ; //storing the value  while(temp!= nullptr) //checking if the temp is not null  {  if(value< temp->data) //then checking the value of temp is less than the temp->data or not  value= temp->data ; //if yess then store the value of data in temp in the value  temp= temp->nextPtr ; //interating  }  while(value>0) //if value is greater than 0  {  value = value/10; //then dividing by 10  digit++; //incrementing digit  }  return digit ; //returning the value of digit  }  static int power(int value , int key)  {  int exp = 1; //taking exp as 1  while(key--) //while key is not zero  {  exp = exp\*value ; //multiplying the exp by that value and storing in exp  }  return exp ; //returning exp  }  static void radixSort(linkList & l , linkList r[])  {  int i = 1;  int toDelete; //declaring  int count1;  int value = max(l); //storing the value got from the function in max  while(i<=value) //checking if the value is greater than 1 or not  {  while(!l.isEmpty()) //then checking if it is not empty  {  toDelete = l.deleteAtStart();  count1 = toDelete%power(10,i); //storing the value  count1 = count1/power(10,i-1); //by decrementing the i storing in count 1  r[count1].append(toDelete); //then appending  }  for(int j=0;j<10;j++) //loop till 10 bcz the array size is 10  {  while(!r[j].isEmpty())  {  toDelete = r[j].deleteAtStart();  l.append(toDelete); //calling by passing to delete  }  }  i++; //incrementing  }  } };  **Main.cpp:**  #include <iostream> #include "Functions.h"  using namespace std;  int main() {  Node n(5); //making object  n.setNextPtr(0);  n.getNextPtr(); //calling  linkList l1; //making object  l1.display();  int opt1; //declaring  do {   cout << "\nCHOOSE OPTIONS." << endl;  cout << "1.APPEND." << endl;  cout << "2.PREPEND." << endl;  cout << "3.ADD AT SPECIFIC INDEX." << endl; //displaying options  cout << "4.DELETE AN ELEMENT FROM THE START OF THE LIST." << endl;  cout << "5.DELETE AN ELEMENT FROM THE LAST OF THE LIST." << endl;  cout << "6.DELETE AN ELEMENT FROM THE SPECIFIC INDEX OF THE LIST." << endl;  cout << "7.QUICK SORT." << endl;  cout << "8.RADIX SORT." << endl;  cout << "9.EXIT." << endl;  cin >> opt1;  if (opt1 == 1) {  l1.append(3);  l1.append(7);  l1.append(2);  l1.append(0);  l1.display();  }   if (opt1 == 2) {  l1.prepend(4);  l1.prepend(0);  l1.prepend(1);  l1.prepend(2);  l1.display();  }  if (opt1 == 3) {  int index;  int value;  cout << "ENTER THE INDEX AT WHICH YOU WANT TO ADD THE VALUE." << endl;  cin >> index;  cout << "ENTER THE NUMBER YOU WANT TO ADD IN LIST = " << endl;  cin >> value; //taking value  l1.insertAtIndex(index, value); //calling  l1.display();  }  if (opt1 == 4) {  l1.deleteAtStart(); //calling  l1.display();  }  if (opt1 == 5) {  l1.deleteAtEnd(); //calling  l1.display();  }  if (opt1 == 6) {  int index;  cout << "ENTER THE INDEX FROM WHICH YOU WANT TO DELETE THE VALUE." << endl;  cin >> index;  l1.deleteAtIndex(index);  l1.display(); //calling  }  if (opt1 == 7) {  Node \*temp = l1.getHead();  cout<<"BEFORE SORTING = ";  l1.display();  l1.quicksort(temp, l1.getTail(temp)); //calling  cout<<"\nAFTER SORTING = ";  l1.display();  }  if (opt1 == 8) {  linkList r[10];  cout<<"BEFORE SORTING = ";  l1.display(); //calling  l1.radixSort(l1,r);  cout<<"\nAFTER SORTING = ";  l1.display();  }  if (opt1 == 9) {  cout << "YOU CHOOSE TO EXIT." << endl; //displaying  exit(3);  }  } while (opt1 >= 1 && opt1 <= 7); //condition to terminate the do while loop   return 0; } |