Implementation of Doubly linked list

```
// code
#include <stdio.h>
#include <stdlib.h>
// Implementation of Doubly linked list

//Declaration of node
struct node {
   int data;
   struct node *previous;
   struct node *next;
};

// Declararion of start of linked list
struct node *start = NULL;

// Second linked list for merging
struct nodeTwo {    // Declaration for secondary linked list
   int dataTwo;
   struct nodeTwo *previousTwo;
   struct nodeTwo *nextTwo;
};
// Start node of secondary linked list
struct nodeTwo *startTwo = NULL;

void secondLinkedList() {     // Initialises second linked list with static values
   // declare nodes
   struct nodeTwo *newNodeOne;
   struct nodeTwo *newNodeTwo;
   struct nodeTwo *newNodeThree;
   // allocates memory for nodes
   newNodeOne = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeTwo = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeThree = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   // enter data and link the nodes
   startTwo = newNodeOne;
   newNodeOne->dataTwo = 4;
   newNodeOne->nextTwo = newNodeTwo;
   newNodeOne->previousTwo = NULL;

   newNodeTwo->dataTwo = 8;
   newNodeTwo->nextTwo = newNodeThree;
   newNodeTwo->previousTwo = newNodeOne;

   newNodeThree->dataTwo = 12;
   newNodeThree->nextTwo = NULL;
   newNodeThree->previousTwo = newNodeTwo;

}

void insertAtBeginning(int toInsert) {    // Inserts at the beginning of the node
```

```
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (start == NULL) {    // first node of is added
        newNode->next = NULL;
        newNode->previous = NULL;
        start = newNode;
    } else {
        // linking newNode before current start
        newNode->next = start;
        newNode->previous = NULL;
        start->previous = newNode;

        // shifting start
        start = newNode;
    }
}

void insertAtEnd(int toInsert) {    // Inserts at the end of the list

    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    if (start == NULL) {    // first node is to be added
        newNode->next = NULL;
        newNode->previous = NULL;
        start = newNode;
    } else {    // any other node
        while (ptr->next != NULL) {    // traverse upto currnet last node
            ptr = ptr->next;
        }
        // link current last node with newNode
        ptr->next = newNode;
        newNode->previous = ptr;
        newNode->next = NULL;
    }
}

void insertBeforeVal(int toInsert, int val) {    // Inserts node before val is encountered

    if (start == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }
```

```
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traverse until val is encountered
        ptr = ptr->next;
    }

    if (ptr->previous == NULL) {    // inserting before current first node
        // linking new node with current first node
        newNode->next = ptr;
        newNode->previous = NULL;
        ptr->previous = newNode;
        // shifting start
        start = newNode;
    } else {
        // linking nodes before val
        newNode->next = ptr;
        newNode->previous = ptr->previous;
        ptr->previous->next = newNode;
        ptr->previous = newNode;
    }

}

void insertAfterVal(int toInsert, int val) {    // Inserts node after val is encountered

    if (start == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traverse until val is encountered
        ptr = ptr->next;
    }

    if (ptr->next == NULL) {    // inserting after current last node
        // linking new node with current last
        ptr->next = newNode;
```

```
        newNode->previous = ptr;
        newNode->next = NULL;
    } else {
        // linking nodes
        newNode->previous = ptr;
        newNode->next = ptr->next;
        ptr->next->previous = newNode;
        ptr->next = newNode;
    }

}

void insertAtPosition(int toInsert, int pos) {    // Inserts node at the given position

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    int count = 1;

    while (count != pos && ptr->next != NULL) {    // traverse list upto position
        ptr = ptr->next;
        count++;
    }

    if (pos > count+1 || pos <= 0) {    // invalid position
        printf("\nList is not that long!");
        return;
    }

    if (count == 1) {    // inserting at first position
        // linking new node with current first node
        newNode->next = ptr;
        newNode->previous = NULL;
        ptr->previous = newNode;
        // shifting start
        start = newNode;
    } else if (ptr->next == NULL && count < pos) {    // inserting at last position
        // linking new node with current last node
        ptr->next = newNode;
        newNode->previous = ptr;
        newNode->next = NULL;
    } else {    // inserting at any position
        newNode->next = ptr;
```

```
        newNode->previous = ptr->previous;
        ptr->previous->next = newNode;
        ptr->previous = newNode;
    }

}

void deleteAtBeginning() {    // deletes at the beginning

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->next == NULL) {    // only remaining node is to be deleted
        start = NULL;
    } else {
        ptr->next->previous = NULL;
        start = ptr->next;
    }
    free(ptr);

}

void deleteAtEnd() {    // deletes at end

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->next != NULL) {    // traversing upto last node
        ptr = ptr->next;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (start->next == NULL) {    // only remaining node is to be deleted
        start = NULL;
    } else {
        ptr->previous->next = NULL;
    }
    free(ptr);

}

void deleteBeforeVal(int val) {    // deletes node before val is encountered
```

```
    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    if (start->data == val) {    // check for invalid input
        printf("\nNo elements before  %d", val);
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->next->data != val) {    // traversing upto last node
        ptr = ptr->next;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->previous == NULL) {    // deleting current first node
        ptr->next->previous = NULL;
        start = ptr->next;
    } else {
        ptr->previous->next = ptr->next;
        ptr->next->previous = ptr->previous;
    }
    free(ptr);

}

void deleteAfterVal(int val) {    // deletes node after val is encountered

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traversing until val is encountered
        ptr = ptr->next;
    }
    if (ptr->next == NULL) {    // check for invalid input
        printf("\nNo elements after  %d", val);
        return;
    }
    // set ptr to node which is to be deleted
    ptr = ptr->next;

    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->next == NULL) {    // deleting current last node
        ptr->previous->next = NULL;
    } else {
        ptr->previous->next = ptr->next;
```

```
        ptr->next->previous = ptr->previous;
    }
    free(ptr);

}

void deleteAtPosition(int pos) {    // deletes at given position

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing node
    struct node *ptr = start;
    int count = 1;

    while (count != pos && ptr->next != NULL) {    // traversing until val is encountered
        ptr = ptr->next;
        count++;
    }
    if (pos > count || pos<=0) {    // invalid position
        printf("\nInvalid position!");
        return;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (start->next == NULL) {    // deleting only remaining node
       start = NULL;
    } else if (count == 1) {    // deleting at first position
        ptr->next->previous = NULL;
        start = ptr->next;
    } else if (ptr->next == NULL) {    // deleting at last position
        ptr->previous->next = NULL;
    } else {    // deleting at any position
        ptr->previous->next = ptr->next;
        ptr->next->previous = ptr->previous;
    }

}

void updateAtBeginning(int toUpdate) {    // updates at the beginning of the list

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // updation of value
    start->data = toUpdate;
}

void updateAtEnd(int toUpdate) {    // updates at the end of the list
```

```
    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->next != NULL) {
        ptr = ptr->next;
    }
    // updation of value
    ptr->data = toUpdate;
}

void updateBeforeVal(int toUpdate, int val) {    // updates the node before val

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    if (start->data == val) {
        printf("\nNo nodes before entered value");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->next->data != val) {
        ptr = ptr->next;
    }
    // updation of value;
    ptr->data = toUpdate;
}

void updateAfterVal(int toUpdate, int val) {    // updates the node after val

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->data != val) {
        ptr = ptr->next;
    }
    if (ptr->next == NULL) {
        printf("\nNo nodes after entered val!");
        return;
    }
    // shifting the pointer to node which is to be updated
    ptr = ptr->next;
    //updation
    ptr->data = toUpdate;
}
```

```
void updateAtPosition(int toUpdate, int pos) {    // updates value at enetered position

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    while (count != pos && ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    if (pos > count || pos<=0) {    // invalid position
        printf("\nInvalid position!");
        return;
    }
    // updation
    ptr->data = toUpdate;
}

int countNodes() {    // Counts number of nodes in the list

    if (start == NULL) {    // if the list is empty
        return 0;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    // traversing
    while (ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    return count;

}

void search(int val) {    // Search weather the val is present in the list and prints its position

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    // traversing
```

```
    while ((count!=countNodes()+1) && (ptr->data != val)) {
        ptr = ptr->next;
        count++;
    }
    // printing
    if (count > countNodes()) {
        printf("\n%d is not present in the list!", val);
    } else {
        printf("\nPosition of %d in the list is :  %d", val, count);
    }
}

void sort() {    // Sorts the list

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }
    struct node *i = start;
    struct node *j = NULL;
    int temp;
    for (i = start ; i != NULL ;  i=i->next) {
        for (j = i->next ; j != NULL ; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

void reverse() {    // Reverses the list

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    struct node *previousNode, *currentNode, *nextNode;
    previousNode = NULL;
    currentNode = nextNode = start;
    while (nextNode != NULL) {
        nextNode = nextNode->next;
        currentNode->next = previousNode;
        currentNode->previous = nextNode;
        previousNode = currentNode;
        currentNode = nextNode;
    }
    start = previousNode;

}
```

```
void merge() {
   struct node *ptr;
   struct nodeTwo *ptrTwo;
   secondLinkedList();
   ptr = start;
   while (ptr->next != NULL) {
      ptr = ptr->next;
   }
   ptr->next = (struct node *)startTwo;
   startTwo->previousTwo = (struct nodeTwo*)ptr;
   sort();
}

void display() {    // Displays elements of the list

   if (start == NULL) {    // check if list is empty
      printf("\nList is empty!");
      return;
   }
   // traversing pointer
   struct node *ptr = start;

   printf("Elements in the list are  : ");
   while (ptr->next != NULL) {
      printf("%d  ", ptr->data);
      ptr = ptr->next;
   }
   printf("%d", ptr->data);

}

void displayListTwo() {

   struct nodeTwo* ptr;
   ptr = startTwo;
   if (ptr == NULL) {
      printf("\nList is empty!");
      return;
   }
   printf("\n");
   while (ptr->nextTwo != NULL) {
      printf("%d  ", ptr->dataTwo);
      ptr = ptr->nextTwo;
   }
   printf("%d ", ptr->dataTwo);

}


int main() {
```

```
int choice, toInsert, toUpdate, val, pos;

while (1) {

    printf("\n*1   INSERT At END  ");
    printf("\n*2   INSERT At BEGINING  ");
    printf("\n*3   INSERT BEFORE VAL  ");
    printf("\n*4   INSERT AFTER VAL  ");
    printf("\n*5   INSERT At POSITION  ");
    printf("\n*6   DELETE At END  ");
    printf("\n*7   DELETE At BEGINING  ");
    printf("\n*8   DELETE BEFORE VAL  ");
    printf("\n*9   DELETE AFTER VAL  ");
    printf("\n*10  DELETE At POSITION  ");
    printf("\n*11  UPDATE At END  ");
    printf("\n*12  UPDATE At BEGINING  ");
    printf("\n*13  UPDATE BEFORE VAL  ");
    printf("\n*14  UPDATE AFTER VAL  ");
    printf("\n*15  UPDATE At POSITION  ");
    printf("\n*16  SEARCH in the list  ");
    printf("\n*17  COUNT NODE in the list  ");
    printf("\n*18  DISPLAY elements of the list  ");
    printf("\n*19  REVERSE List  ");
    printf("\n*20  SORT List");
    printf("\n*21  MERGE List");
    printf("\n*22  EXIT  ");
    printf("\nEnter your choice :  ");
    scanf("%d", &choice);

    switch (choice) {

        case 1:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            insertAtEnd(toInsert);
            break;

        case 2:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            insertAtBeginning(toInsert);
            break;

        case 3:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            printf("\nEnter value BEFORE which to insert :  ");
            scanf("%d", &val);
            insertBeforeVal(toInsert, val);
            break;

        case 4:
```

```
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter value AFTER which to insert :  ");
    scanf("%d", &val);
    insertAfterVal(toInsert, val);
    break;

case 5:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter POSITION AT which to insert :  ");
    scanf("%d", &pos);
    insertAtPosition(toInsert, pos);
    break;

case 6:
    deleteAtEnd();
    break;

case 7:
    deleteAtBeginning();
    break;

case 8:
    printf("\nEnter value BEFORE which to DELETE :  ");
    scanf("%d", &val);
    deleteBeforeVal(val);
    break;

case 9:
    printf("\nEnter value AFTER which to DELETE :  ");
    scanf("%d", &val);
    deleteAfterVal(val);
    break;

case 10:
    printf("\nEnter POSITION AT which to DELETE :  ");
    scanf("%d", &pos);
    deleteAtPosition(pos);
    break;

case 11:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtEnd(toUpdate);
    break;

case 12:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtBeginning(toUpdate);
    break;
```

```
case 13:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter value BEFORE which to UPDATE :  ");
   scanf("%d", &val);
   updateBeforeVal(toUpdate, val);
   break;

case 14:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter value AFTER which to UPDATE :  ");
   scanf("%d", &val);
   updateBeforeVal(toUpdate, val);
   break;

case 15:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter POSITION AT which to UPDATE :  ");
   scanf("%d", &pos);
   updateAtPosition(toUpdate, pos);
   break;

case 16:
   printf("\nEnter a value to SEARCH :  ");
   scanf("%d", &val);
   search(val);
   break;

case 17:
   printf("\nList contains  %d  elements", countNodes());
   break;

case 18:
   printf("\nElements in the list are :  ");
   display();
   break;

case 19:
   reverse();
   printf("\nList is reversed");
   break;

case 20:
   sort();
   printf("\nList is sorted");
   break;

case 21:
   merge();
```

```
            printf("\nTwo lists are merged!");
            break;

        case 22:
            printf("*** E X I T I N G ***");
            exit(1);
            break;

        default:
            printf("INVALID INPUT");
      }

   }

   return 0;
}
```

// output

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  1

Enter element to insert :  5

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
```

```
Enter your choice :  1

Enter element to insert :  10

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  1

Enter element to insert :  15

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
```

```
Enter your choice :  18
Elements in the list are  : 5  10  15
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  11

Enter element to UPDATE :  200

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
```

```
Enter your choice :  18
Elements in the list are  : 5  10  200
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  9

Enter value AFTER which to DELETE :  10

Deleted element is :  200
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
```

```
Enter your choice :  18
Elements in the list are  : 5  10
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  22
*** E X I T I N G ***
Process returned 1 (0x1)   execution time : 48.254 s
Press any key to continue.
```