```
char* partialCopyString(char* str, bool copy_even, int* out_len) {
//BAD CODING : not initiallize to NULL
        char* OUT;
  if (copy_even) {
//CORECTNESS : if we allocate memory with a valid pointer we loose it
        out_len = malloc(sizeof(int));
//CORECTNESS : if strlen(str) == 7 for example out_len will recive 3 and not 4
        *out_len = strlen(str) / 2;
//BAD CODING : in all operating systems char = 1 bype but still i would write it
//BAD CODING : needs to check if memory allocation failed
//CORECTNESS : for new string we need to allocate memory of [strlen(str) + 1]
    OUT = malloc(*out_len);
//CORECTNESS : we will read from a non allocate memory in last iteration
    for (int i = 0; i < strlen(str); i += 2) {
        OUT[i / 2] = str[i + 1];
    }
  }else{
//CORECTNESS : if we allocate memory with a valid pointer we loose it
        out_len = malloc(sizeof(int));
        *out_len = strlen(str) / 2 + strlen(str) % 2;
//BAD CODING : in all operating systems char = 1 bype but still i would write it
//BAD CODING : needs to check if memory allocation failed
//CORECTNESS : for new string we need to allocate memory of [strlen(str) + 1]
    OUT = malloc(*out_len);
    for (int i = 0; i < strlen(str); i += 2)  {
        OUT[i / 2] = str[i];
    }
  }
  return OUT;
}
```

```
char* partialCopyString(char* str, bool copy_even, int* out_len) {
       char* OUT = NULL;
       if (copy_even) {
             *out_len = 0;
        OUT = malloc( (*out_len + 1) * sizeof(*OUT));
        if (OUT == NULL)  return NULL;

        for (int i = 0; i < strlen(str); i++) {
             if (i % 2 == 1) {
                    OUT[i/2] = str[i];
                    *out_len = *out_len + 1;
             }
        }
        OUT[*out_len] = '\0';
       }else{
             *out_len = strlen(str) / 2 + strlen(str) % 2;
        OUT = malloc( (*out_len + 1) * sizeof(*OUT));
        for (int i = 0; i < strlen(str); i += 2) {
             OUT[i / 2] = str[i];
        }
        OUT[*out_len] = '\0';
       }
    return OUT;
}
```

```
//=======================================================================
//the function create an element , insert data into it and return its
//pointer
//element->next recive NULL
//return NULL if memory allocation failed
Node createElement(int data) {

        Node element = malloc( sizeof(*element) );
        if (element == NULL)      return NULL;

        element->data = data;
        element->next = NULL;

        return element;
}
//-----------------------------------------------------------------------
//copy element and returns its pointer
Node copyElement(Node element) {

        Node element_copy = malloc( sizeof(*element_copy) );
        if (element_copy == NULL) return NULL;

        element_copy->data = element->data;
        element_copy->next = element->next;

        return element_copy;
}
//-----------------------------------------------------------------------
//release all the memory allocated for list
void listDestroy(Node list) {

        Node temp = NULL;

        while (list != NULL) {
                temp = list->next;
                free(list);
                list = temp;
        }
}
//-----------------------------------------------------------------------
//add element to the list in original order
//return false if memory allocation failed and true otherwise
bool addToListDirect(Node* list_ptr,const Node const element) {

        Node element_copy = copyElement(element);
        if (element_copy == NULL) return false;

        if (*list_ptr == NULL) {
                *list_ptr = element_copy;
                return true;
        }

        Node list_running = *list_ptr;
        while (list_running->next != NULL) {
                list_running = list_running->next;
        }
        list_running->next = element_copy;

        return true;
}
```

```
//-------------------------------------------------------------------------
//add element to the list in a reversed order
//return false if memory allocation failed and true otherwise
bool addToListRevers(Node* list_ptr,const Node const element) {

        Node element_copy = copyElement(element);
        if (element_copy == NULL) return false;

        element_copy->next = *list_ptr;
        *list_ptr = element_copy;

        return true;
}
//-------------------------------------------------------------------------
//printing all the elements of a given list
void printList(const Node const list) {

        Node list_running = list;
        while (list_running != NULL) {
                printf("%d ",list_running->data);
                list_running = list_running->next;
        }
}
//-------------------------------------------------------------------------
//build a a list from element in even places and a reversed list from
element
//placed in odd places
//prints the list and then the reversed list
bool printCheckmarkOrdered(const Node const list) {
        if (list == NULL)   return true;
        Node list_running = list;
        int counter = 0;
        Node list_direct = NULL;
        Node list_revers = NULL;
        Node element = NULL;
        bool result;
        while (list_running != NULL) {
                element = createElement(list_running->data);
                if (element == NULL)      return false;
                if (counter % 2 == 0) {
                        result = addToListDirect(&list_direct,element);
                }else{
                        result = addToListRevers(&list_revers,element);
                }
                if (result == false) {
                        free(element);
                        listDestroy(list_direct);
                        listDestroy(list_revers);
                        return false;
                }
                counter = counter + 1;
                list_running = list_running->next;
                free(element);
        }
        printList(list_direct);
        printList(list_revers);
        listDestroy(list_direct);
        listDestroy(list_revers);
        return true;
}
//=========================================================================
```