**Q1-errors:**

**char**\* **partialCopyString**(**char**\* str, bool copy\_even, **int**\* out\_len) {

**char**\* OUT; *//BAD CODING : variables with uppercast letters*

*// pointer not initiallized to NULL*

**if** (copy\_even) {

out\_len = **malloc**(**sizeof**(**int**));*//BAD CODDING : no need to reallocate*

*memory for out\_len*

*and even if we wanted*

*to we would do it without malloc*

\*out\_len = **strlen**(str) / 2;

OUT = **malloc**(\*out\_len);*//CORECTNESS : should be*

*// malloc((\*out\_len + 1)\*sizeof(\*OUT)).*

*//CORRECTNESS : there is no check if malloc*

*succeed*

**for** (**int** i = 0; i < **strlen**(str); i += 2) {

OUT[i / 2] = str[i + 1];

*//EXPLANATION : working well only for odd length of str because the \0 in //even lengths is positioned in even index but* *the copy is only from odd //indexes as those are the indexes of* *counted even. therefore there is no //copy of \0 for some* *inputs. to get the minimal change we offered below a //solution, but this could have been better with better logic structure.*

*//for example: plan it diffrent by cases of the parity of strlen(str).*

}

} **else** {

out\_len = **malloc**(**sizeof**(**int**));*//same as before*

\*out\_len = **strlen**(str) / 2 + **strlen**(str) % 2;

OUT = **malloc**(\*out\_len);*//same as before*

**for** (**int** i = 0; i < **strlen**(str); i += 2) { *//CORECTNESS : wont*

*copy '\0'.*

OUT[i / 2] = str[i];

}

}

**return** OUT;*//CORECTNESS : the function didn't freed the memory*

*allocated for OUT*

}

**Q1-Corect:**

**char**\* **partialCopyString**(**char**\* str, bool copy\_even, **int**\* out\_len) {

**char**\* out = NULL;

**if** (copy\_even) {

\*out\_len = **strlen**(str) / 2;

out = **malloc**((\*out\_len + 1)\*(**sizeof**(\*out)));

**if**(!out) **return** NULL;

**for** (**int** i = 0; i < **strlen**(str); i += 2) {

out[i / 2] = str[i + 1];

}

} **else** {

\*out\_len = **strlen**(str) / 2 + **strlen**(str) % 2;

out = **malloc**((\*out\_len + 1)\*(**sizeof**(\*out)));

**if**(!out) **return** NULL;

**for** (**int** i = 0; i < **strlen**(str); i += 2) {

out[i / 2] = str[i];

}

}

out[\*out\_len] = '\0';

**return** out;

}

**Q2:**

*//=========================================================================*

**typedef** **enum** {

***PRINT\_AND\_DESTROY***,

***DESTROY***

} **Action**;

*//-------------------------------------------------------------------------*

*//the function create a node , insert data into it and return its pointer.*

*//node->next recive NULL.*

*//return NULL if memory allocation failed.*

**Node** **nodeCreate**(**int** data) {

**Node** node = **malloc**( **sizeof**(\*node) );

**if** (node == NULL) **return** NULL;

node->*data* = data;

node->*next* = NULL;

**return** node;

}

*//-------------------------------------------------------------------------*

*//the function destroys a node.*

**void** **nodeDestroy**(**Node** node){

**free**(node);

}

*//-------------------------------------------------------------------------*

*//the function copy a node and returns its pointer.*

*//return NULL if memory allocation failed*

**Node** **nodeCopy**(**Node** node) {

**Node** copy = **malloc**( **sizeof**(\*copy) );

**if** (copy == NULL) **return** NULL;

copy->*data* = node->*data*;

copy->*next* = node->*next*;

**return** copy;

}

*//-------------------------------------------------------------------------*

*//release all the memory allocated for list.*

**void** **listDestroy**(**Node** head) {

**Node** temp = NULL;

**while** (head != NULL) {

temp = head->*next*;

**nodeDestroy**(head);

head = temp;

}

}

*//-------------------------------------------------------------------------*

*//add node to the list in original order.*

*//return false if memory allocation failed and true otherwise.*

bool **addToListDirect**(**Node**\* head\_ptr,**const** **Node** **const** node) {

**Node** copy = **nodeCopy**(node);

**if** (copy == NULL) **return** false;

**if** (\*head\_ptr == NULL) {

\*head\_ptr = copy;

**return** true;

}

**Node** list\_running = \*head\_ptr;

**while** (list\_running->*next* != NULL) {

list\_running = list\_running->*next*;

}

list\_running->*next* = copy;

**return** true;

}

*//-------------------------------------------------------------------------*

*//add node to the list in a reversed order*

*//return false if memory allocation failed and true otherwise*

bool **addToListRevers**(**Node**\* head\_ptr,**const** **Node** **const** node) {

**Node** node\_copy = **nodeCopy**(node);

**if** (node\_copy == NULL) **return** false;

node\_copy->*next* = \*head\_ptr;

\*head\_ptr = node\_copy;

**return** true;

}

*//-------------------------------------------------------------------------*

*//by its received action argument, the function prints and calls //listDestroy or* *only destroy. the destroy fields are either two lists and //a node or just* *two lists(which is being decided by the first input //argument).*

**void** **makeAction**(**const** **Node** **const** node,**const** **Node** **const** first\_list,

**const** **Node** **const** second\_list, **Action** action) {

**if**(action == ***PRINT\_AND\_DESTROY***){

**Node** list\_running;

**for**(**int** i = 0; i < 2; i++){

**if**(i == 0){

list\_running = first\_list;

} **else** {

list\_running = second\_list;

}

**while** (list\_running != NULL) {

**printf**("%d ",list\_running->*data*);

list\_running = list\_running->*next*;

}

}

}

**if**(node != NULL) **nodeDestroy**(node);

**listDestroy**(first\_list);

**listDestroy**(second\_list);

}

*//-------------------------------------------------------------------------*

*//build a a list from node in even places and a reversed list from node*

*//placed in odd places*

*//prints the head and then the reversed list*

bool **printCheckmarkOrdered**(**const** **Node** **const** head) {

**if** (head == NULL) **return** true;

**Node** list\_running = head;

**Node** list\_direct = NULL;

**Node** list\_revers = NULL;

**Node** node = NULL;

**int** counter = 0;

bool result;

**while** (list\_running != NULL) {

node = **nodeCreate**(list\_running->*data*);

**if** (node == NULL){

**makeAction**(NULL, list\_direct, list\_revers, ***DESTROY***);

**return** false;

}

**if** (counter++ % 2 == 0) {

result = **addToListDirect**(&list\_direct,node);

}**else**{

result = **addToListRevers**(&list\_revers,node);

}

**if** (result == false) {

**makeAction**(node, list\_direct, list\_revers, ***DESTROY***);

**return** false;

}

list\_running = list\_running->*next*;

**nodeDestroy**(node);

}

**makeAction**(NULL, list\_direct, list\_revers, ***PRINT\_AND\_DESTROY***);

**return** true;

}

*//=========================================================================*