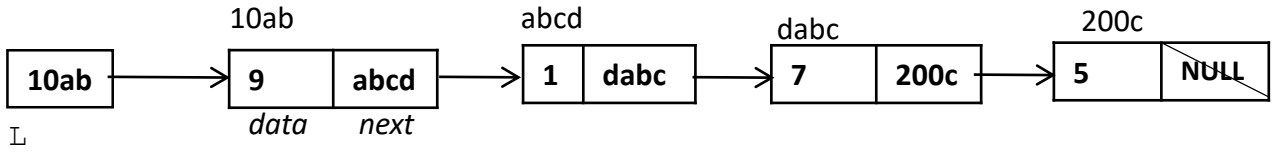
C review
Single Linked Lists
Dynamic Memory

Alexandra Stefan

Outline

- Brief C discussion: malloc/calloc/free
- Static vs Dynamically allocated memory
- Drawing nodes and pointers
- "Cheat sheet" for linked lists
- Code - solution and drawing for:
  - Create a list from an array (`array_2_list(…)`), delete an entire list (`destroy_list(…)`)
  - Insert/dele a node after a node and from a list
  - Swap two consecutive nodes in a list
- Array of linked list – example and drawing
- Steps for developing the solution and the code for problems that involve loops
- Steps for `array_2_list(…)`
- Program state at different times for `array_2_list(…)`
- Steps for `destroy_list(…)`
- Worksheets for the problems solved above

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```



Dynamic memory management: malloc()/calloc()/realloc() and free()

- References:
  - https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm
  - https://www.cplusplus.com/reference/cstdlib/malloc/
- malloc() – requests a chunk of memory of a size (in bytes) given as an argument. Returns a pointer (the memory address of the first byte in that chunk) . It returns NULL if failed (if it could not reserve the required amount of memory). This memory must be released with free().
- ***calloc()*** – similar and ***initializes all bits to 0***. Takes number of items and size of an item. It is useful when requesting memory for several items of the same type. E.g. to store an array. This memory must be released with free().
- realloc() resizes the memory. It returns the pointer to the new memory and frees the original. This memory must be released with free().
- **free()** – releases the memory allocated by any one of the allocating method above when given the pointer returned by that method.
- Number of executed CALLS to free() must be EQUAL to the number of executed CALLS to malloc() and calloc(). Otherwise memory leaks occurs.

E.g.

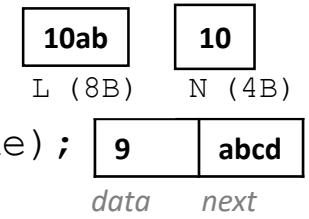
```
nodePT L=NULL;
L = (nodePT)malloc(sizeof(struct node));
free(L);
```

Assume:

```
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Static (S) memory vs Dynamic (D) memory

- Allocated when?
  - S - Allocated *before* the program starts executing.
  - D - Allocated and freed *during* the program execution. Can change size.
  - See "Difference between Static and Dynamic Memory Allocation in C"
- Created how?
  - S - With variable declaration. E.g. `nodePT L;` (or `int n=10;`)
  - D - With malloc()/calloc()/realloc(). E.g. `L=malloc(sizeof(struct node);`

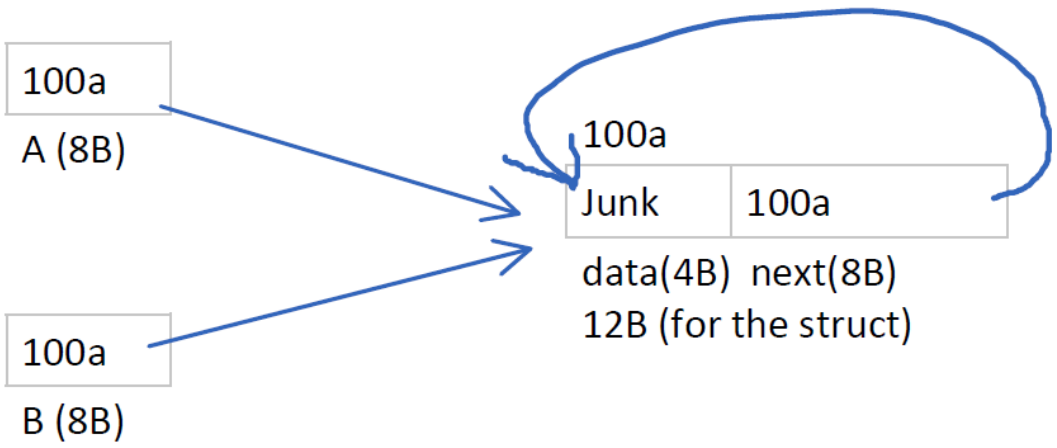


- Freed when?
  - S - When the function call finished (for variables local to that function), or when the program finishes (for global and static variables)
  - D - when free() is called for that pointer.
- Named?
  - S – Yes: Variables (created with variable declaration) are named memory boxes. Using their name we read, or modify the content of that memory box. E.g. `printf("%d",N); N=20; int arr[20];`
  - D – No: Dynamic memory boxes (chunks) are UNNAMED. NO variable NAME is associated with them at creation (and thus have no name). Can be accessed from their pointer. E.g. . `printf("%d",*int_ptr); printf("%d",L->data); L->next=NULL;`
    - Being aware of this difference may avoid confusion.

Static vs Dynamic memory - drawing

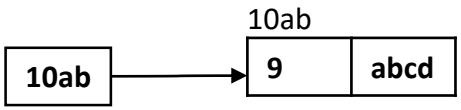
- Below the boxes (the memory) for A and B is static and for the node box it is dynamic
- Also remember that when A=B the content of box labeled B is copied into box labeled A.

```
struct node * A;     //line 1
struct node * B = malloc(sizeof(struct node)); //line 2
A = B;           //line 3
B->next = A;     //line 4
```



Total memory used: 8B+8B+12B=28B

Cheat sheet for Linked Lists



- Dynamically allocated struct vs local pointer variable -
  - DO not confuse the two! Use malloc/calloc to allocate space for a node and then use a POINTER VARIABLE to hold the address of nodes and move through them (just as you use variable j to move through numbers, say 0 to N)
- Must have one malloc/calloc call for every NODE needed.
- CALLS to malloc = CALLS to free
- To delete a node or insert a node, we must know the node that will be BEFORE it – (for single-linked lists)
- Check that any pointer dereferenced is not NULL. (i.e. should never have "NULL->")
- Test cases:
  - L is NULL, L has only one node,
  - Node being worked on is the first node or the last node (e.g. for deletion)
- When swapping, NAME the nodes to avoid overwriting a link
- DRAW the data. MAKE UP values for the memory addresses and any other data needed.
- LOOP to iterate through all the nodes in a list (assuming L points to the first actual node of the list)

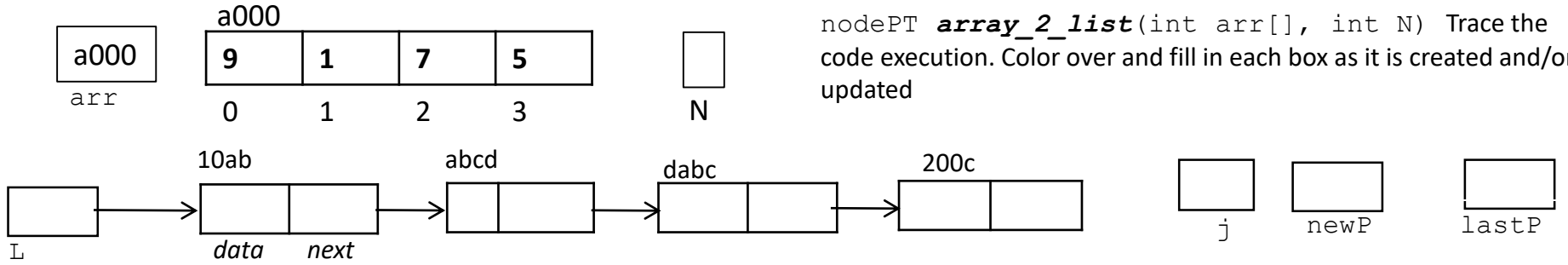
```
for(curr=L; curr!=NULL; curr=curr->next)
```

`curr` is the variable referencing every node (just like j holds numbers 0 to N)

`curr = L` // makes curr point to the first node by holding the mem address of that node (like j=0)

`curr!=NULL;` // this is true when `curr` points to a valid node. When `curr` is the last node, `curr->next` is NULL, thus `curr=curr->next` makes `curr` be NULL

`curr = curr->next` // makes `curr` point to the following node (by holding now the address of that node)



nodePT ***array_2_list***(int arr[], int N) Trace the code execution. Color over and fill in each box as it is created and/or updated

```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {//TC=⊙(N), SC=⊙(1)
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)       {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Note: this code can be written even simpler, but this version is very explicit and can be applied to other scenarios.

Function `array_2_list`
Two implementations

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

For both functions: time: Θ(N), space: Θ(1)

```
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)        {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

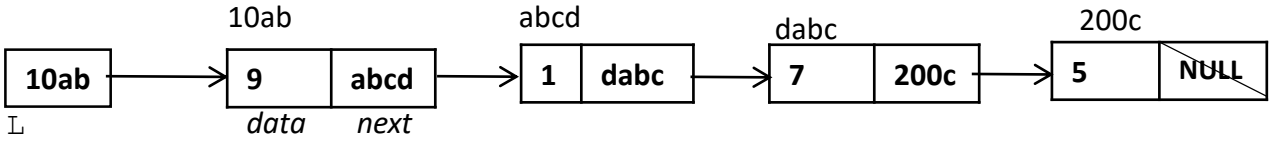
Same code, but use function `new_node()` to create a node.
Advantages:

1. The code is more readable.
2. The new_node() function can be used in other places.
3. If a change (improvement or bug fix) is done new_node(), it is done only once. (No code duplication)

```
nodePT new_node(int value_in) {
  nodePT result =
malloc(sizeof (struct node));
  result->data = value_in;
  result->next = NULL;
  return result;
}
nodePT array_2_list(int arr[], int N)
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = new_node(arr[0]);
  lastP = L;
  for (j = 1; j< N; j++)       {
     newP = new_node(arr[j]);
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

Delete an entire list: `nodePT destroy_list(nodePT L)`

Given data:

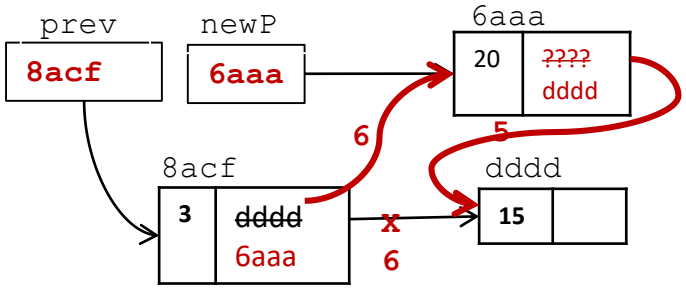


Final data:

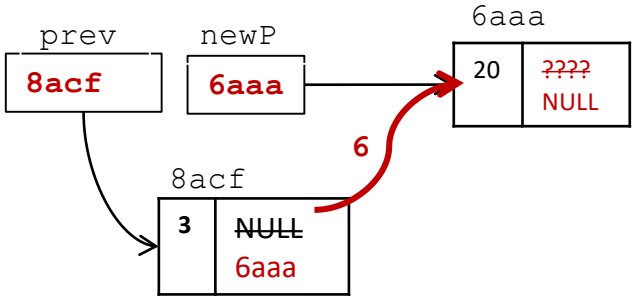

```
// Time complexity: Θ(N),
// where N is the size of the list
nodePT destroy_list(nodePT L) {
    nodePT next, curr;
    curr = L;
    while (curr!=NULL) {
        next = curr->next;
        free(curr);
        curr = next;
    }
    return NULL; // to update pointer in caller code
}
```

Insert a node after a given node – node operation

```
/* Inserts newP after the node "prev".
Note that this is works on nodes. It does not matter how a list is
represented. prev is just a node.   */
void insert_node_after(nodePT prev, nodePT newP) {
    if ((prev == NULL)||(newP==NULL)) {
        printf("\n Cannot insert after a NULL node. No action
taken.");
    } else {
        newP->next = prev->next; //5
        prev->next = newP;  //6
    }
}
```



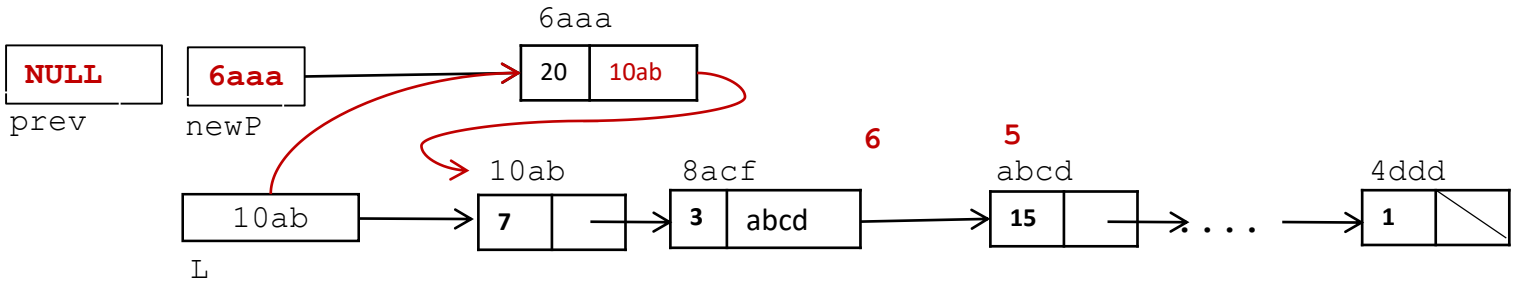
Case when prev->next is NULL works fine: Because we never ACCESS (with ->) the that address, but we only COPY that NULL from one box into another



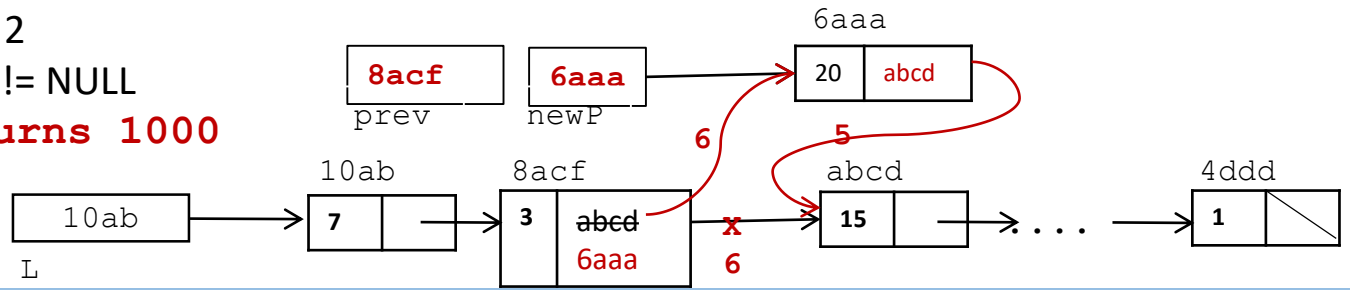
Insert in a list, L, after a given node (assumed from L) Θ(1)

```
/* Inserts in list L, a the new node newP, after the node prev.
 If prev is NULL it means newP must be linked to the beginning of L
 Uses the list representation (L points to the first node with data)  */
nodePT insert_node(nodePT L, nodePT prev, nodePT newP){
   if (prev == NULL) { // case 1: inserts at the beginning of the list L
      newP->next = L;  //2
      return newP;     //3
   }
   else {  // case 2:
      insert_node_after(prev, newP); //4 does not affect the list head
      return L;                      //5
   }
}
```

Case 1:
prev==NULL
Returns 6aaa



Case 2
prev != NULL
Returns 1000

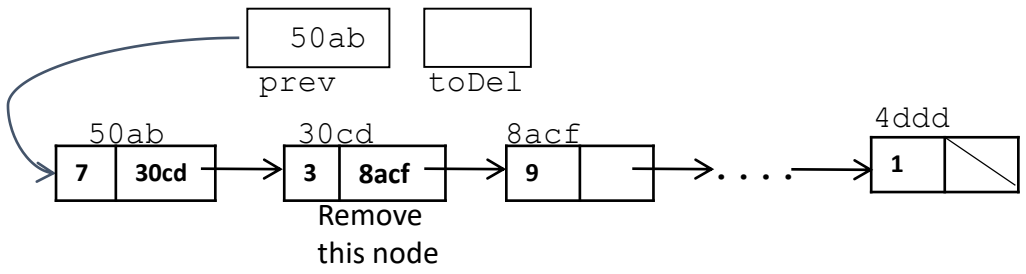


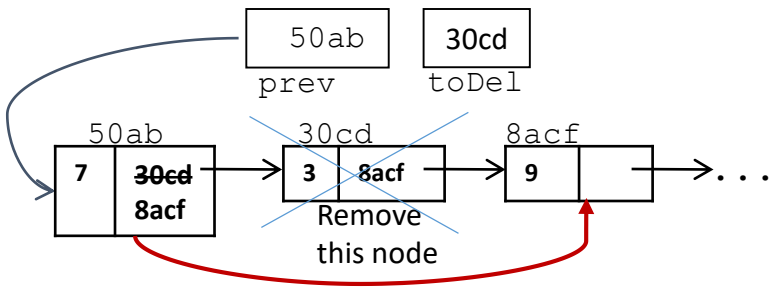
Q: Change the function to not return anything (remove line 5), and replace line 3 with L=newP .Will it work? Will it be correct? What scenario will be best for testing that?

Delete a node after a given node – node operation Θ(1)

```
/* Delete the node after the node "prev".
   Note that this is works on nodes. It does not matter how a list is
represented. prev is just a node.*/
void delete_node_after(nodePT prev) {
    if (prev == NULL) {
        printf("\n Cannot delete after a NULL node. No action taken.");

    } else {
            nodePT toDel = prev->next;   // 3
            if (toDel != NULL){          // 4
                prev->next = toDel->next; // 5 this crashes if toDel is NULL
                free(toDel);             // 6
            }
    }
}
```

SOLUTON drawing:

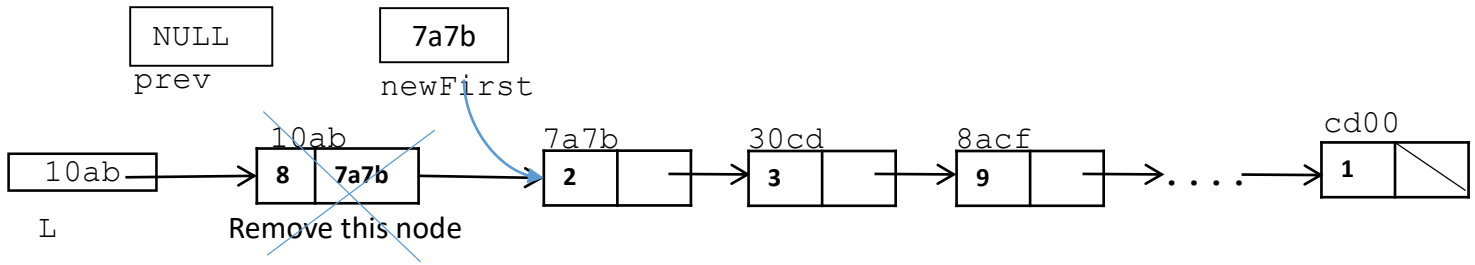




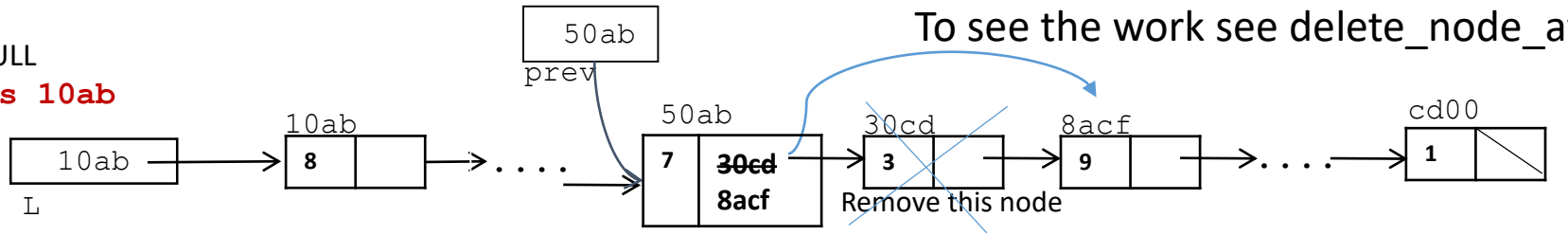
Delete in a list, L, after a given node (assumed from L) - Θ(1)

```
/* Deletes from list L, the node after prev. If prev is NULL it means that the first
node of L must be deleted. Uses the list representation: L points to the 1st node.*/
nodePT delete_node(nodePT L, nodePT prev){
   if (prev == NULL) { // delete the first node from L
      if (L==NULL) {  return NULL; }  // no node in the list. nothing to delete
      else {// case 2: delete 1st node and return the address of the new 1st node
         nodePT newFirst = L->next;
         free(L);
         return newFirst;
       }
   } else {  // case 3
      delete_node_after(prev); // does not affect the list head
      return L;
   }
}
```

Case 2:
prev==NULL, L!=NULL
Returns 7a7b



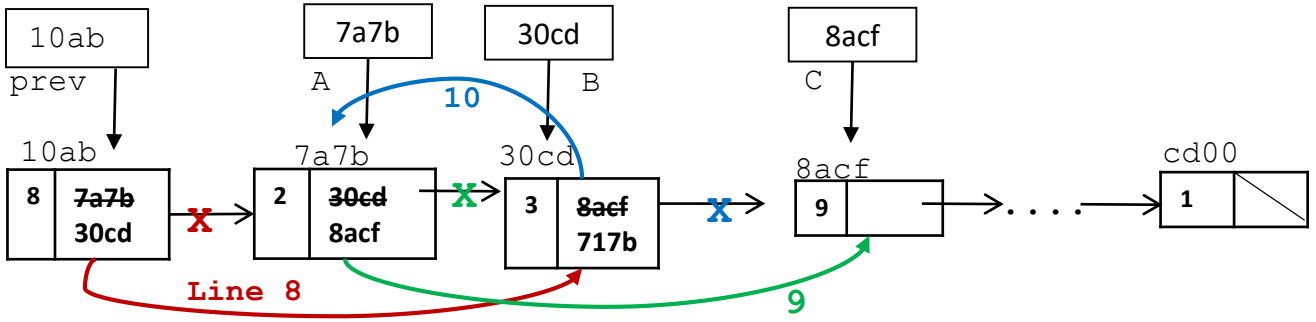
Case 3:
prev!=NULL
Returns 10ab



Swap the next 2 nodes after node prev Θ(1)

HINT: When swapping, NAME the nodes to avoid overwriting a link. Below lines 8,9,10 can be executed in any order. If not named, a specific order would be needed.

```
// Swaps 2 nodes after prev. If prev is NULL or not enough nodes, it does nothing.
void swap_2_after(nodePT prev){
  if ( (prev == NULL) || (prev->next == NULL) || (prev->next->next == NULL) ) {
    printf("\n prev is NULL or not enough nodes!\n");
    return;
  }
  nodePT A = prev->next;  //  1st node in the swap, code crashes if NULL
  nodePT B = prev->next->next;  // 2nd node in the swap, code crashes if NULL
  nodePT C = B->next;  //1st node after the nodes to be swapped (A, B). Ok if NULL
  prev->next = B; //8
  A->next = C;    //9
  B->next = A;    //10
}
```

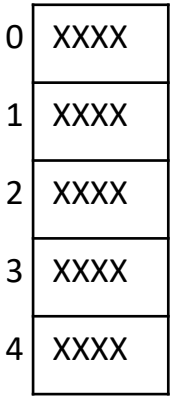


Array of linked lists – simple example

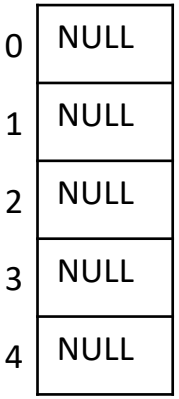
```
/* assume new_node(), array_2_list(), and
print_list_horiz() are the ones from the
list implementation provided.
*/
typedef struct node * nodePT;
truct node {
    int data;
    struct node * next;
};

int arr[] = {5,1,8};
nodePT listArr[5]; //1
// size: 5*sizeof(memory address) = 5*8B=40B

// set every pointer/list to NULL
for(j=0; j<5; j++)  {  // 2
       listArr[j]=NULL;
}
listArr[0] = new_node(5);      //4
listArr[2] = array_2_list(arr, 3); //5
print_list_horiz(listArr[0]);
print_list_horiz(listArr[1]);
print_list_horiz(listArr[2]);
print_list_horiz(listArr[3]);
```

Drawings of listArr at different stages in the program.

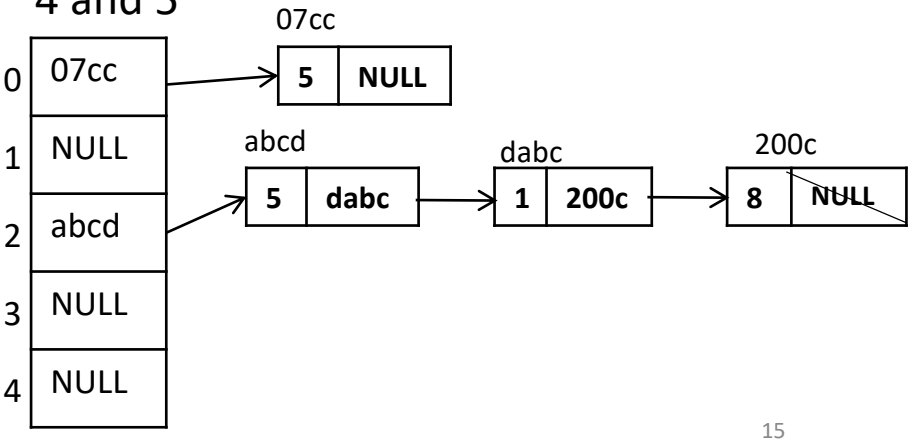
listArr created in line 1



listArr after loop in line 2



listArr after lines 4 and 5



Steps for developing an algorithm (and code) with a loop – (similar to proof by induction)

- Any code that has a loop can only be correct if there is a specific property that the loop preserves. More specifically, there is a relation between the current state of program DATA and the iteration of that loop.
- **0.** When developing code that involves loops, first **draw a picture of the *given data* and the *final resulting data***.

Then ***start form the data*** (the actual data and the variables that you will use to store and access that data) and ***the relation between the data and the loop iteration***.

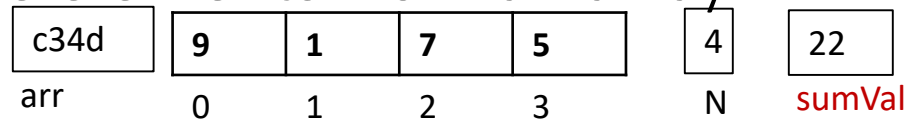
- **1: loop** - decide roughly what the loop does (overall and in one iteration)
- **2a: identify property** - What is the expected **program state before iteration j**. (CLEARLY state what each variable holds: each variable must have a clear meaning and must hold specific data (related to processing the first (j-1) items/data).
- **2b: j -> (j+1)** - **assume** the property holds before iteration j and prove/check it holds before iteration (j+1), i.e. running the code iteration j, preserved that property .

After the current iteration, j, the variables will hold the same information but related to processing the first j items.

- **3: solved in the end** - check that when the loop finished, the problem is solved
- **4: fix start** - check and fix so that the data has the property immediately before the FIRST iteration starts. Most times, this needs fixing.
- PROGRAM STATE = all variables and their content and any other memory or data accessed by the program at THAT SPECIFIC TIME in the execution.
- Below is an example for using this method to compute the sum of the elements in an array of int and
- to create a single linked list with data from an array of int

Steps for developing an algorithm (and code) with a loop – for computing sum over the elements from an array

- `int sumArr(int arr[], int N)`



- **0. Draw a picture of the *given data* and the *final resulting data*.**

Then ***start form the data*** and ***the relation between the data and the loop iteration***.

- **1: loop (& vars)** - decide roughly what the loop does (overall and in one iteration)
  - **At each iteration, add one more number from the array, for(j=0; j<N; j++) {// add arr[j]**
- **2a: identify property** - What is the expected **program state before iteration j**.

Before iteration j, sumVal will have the sum of the elements at indexes 0 to (j-1) , sumVal =sumVal+arr[j]. E.g. for before j=2, sumVal=10

- **2b: j -> (j+1)** - **assume** the property holds before iteration j and prove/check it holds before iteration (j+1), i.e. running the code iteration j, preserved that property .

in iteration for j=2 we do: sumVal=sumVal+arr[j] = 10+arr[2] = 10+7=17 => yes j->(j+1)

- **3: solved in the end** - check that when the loop finished, the problem is solved

Yes, it stops when j is N, i.e. here when j is 4. By case 2 above now sumVal has the sum of elements from indices 0 to N-1 (here indexes: 0,1,2,3) .

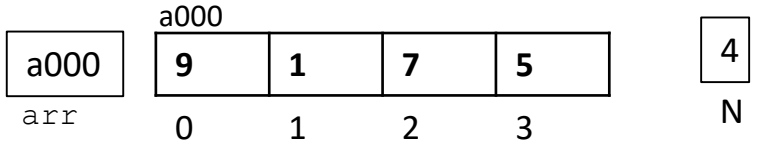
- **4: fix start** - check and fix so that the data has the property immediately before the FIRST iteration starts. Most of the times, this needs fixing.

before iteration for j =0 starts, what is sumVal? It should be 0 => sumVal = 0

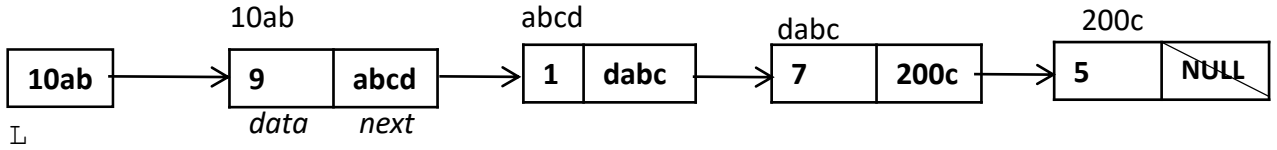
```
int sumArr(int arr[], int N) {
    int j, int sumVal=0;
    for(j=0; j<N j++) { sumVal=sumVal+arr[j]; }
    return sumVal;
}
```

Step 1 - Creating a linked list with data from an array of int

Given data:
array of int, `arr` and int N Drawing:



Data to be created:
Single linked list. Drawing:




```
nodePT array_2_list(int arr[], int N)
```

Solve the problem using the relation between data and loop iteration

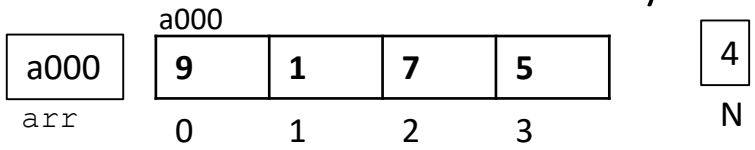
Step 1. What will control the loop? What do we loop over?
Ans: Add a node for one item in arr.

We will iterate over the array `arr`, using the index, j (

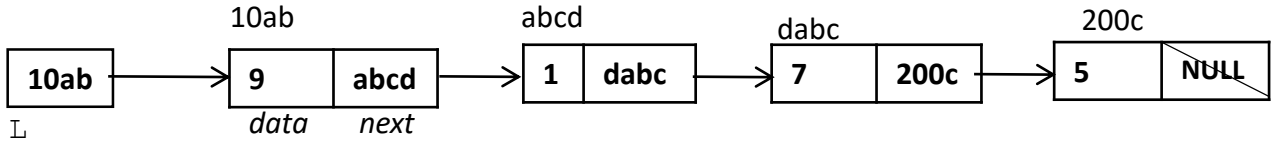
```
for(j=0;j<N;j++) {
        // create a new node,
        // write arr[j] as data in it, (and possibly NULL in next)
        // add it to the end of the list
}
```

Step 2a - Creating a linked list with data from an array of int

Given data:
array of int, `arr` and int N Drawing:



Data to be created:
Single linked list. Drawing:

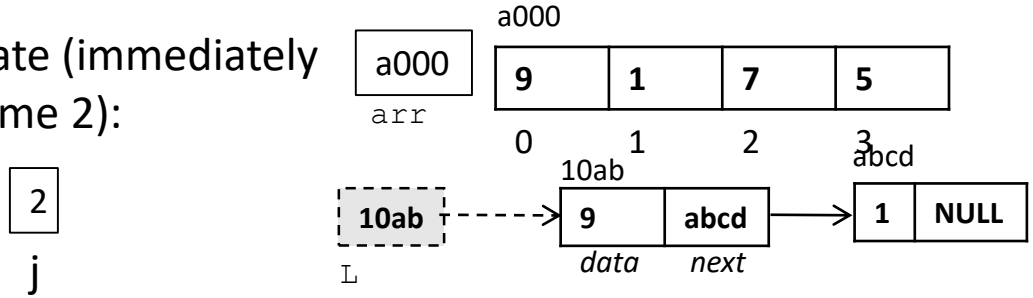


Solve the problem using the relation between data and loop iteration continued

Step 2a: What is the expected program state before iteration j (program state means what value will the variables have).

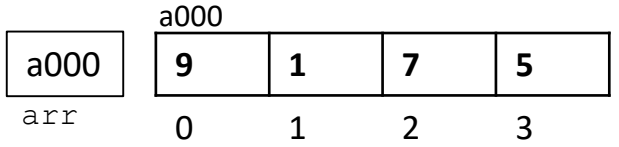
a. Items at indexes 0 to (j-1) were processed, a node was created for each one of them and they are in linked in a linked list in this order. The last node will have arr[j-1] as *data*. E.g. before (j=2) have the nodes at addresses 10ab and abcd, and abcd has *data* 1.
b. What is *next* for the last node (abcd)? Should it be a valid memory address or NULL? I choose NULL so that it is a correct last node in the list and it does not have what to point at

Program state (immediately after j became 2):

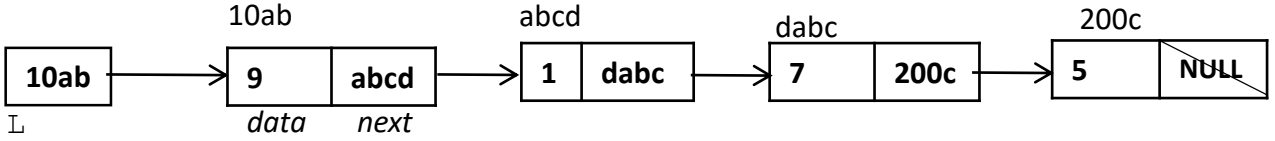


Step 2b - Creating a linked list with data from an array of int

Given data:
array of int, `arr` and int `N` Drawing:



Data to be created:
Single linked list. Drawing:



Solve the problem using the relation between data and loop iteration continued.

Step 2b. *j-> (j+1)* Work done in one iteration (must preserve the state):

What should be done in the iteration when j=2?

a. Create a new node, store its address in a variable*, say* `newP`:

```
struct node * newP = malloc(sizeof(struct node)),
```

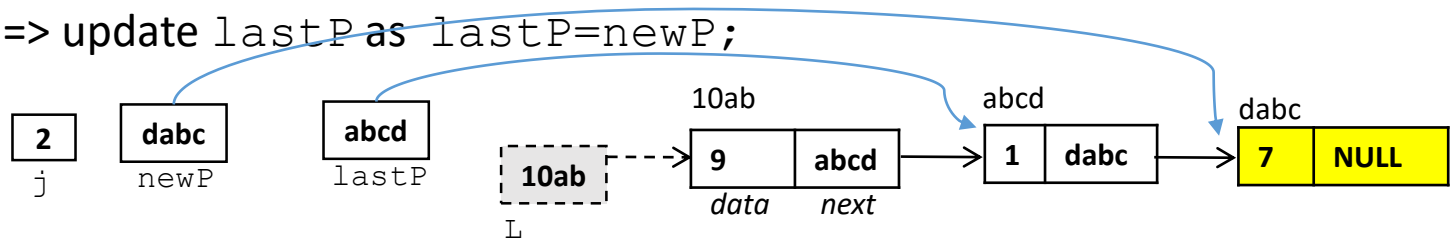
b. ***Write data in it***: copy arr[j] in its *data* field, NULL in *next*

```
newP->data=arr[j]; newP->next=NULL
```

c. **Link the new node at the end of the current list**: set the *next* of the last node in the list (abcd) to have the memory address of the new node. We just realized we need a name for that last node, thus we need a (struct node *) variable. Say `lastP` of type `struct node *` .

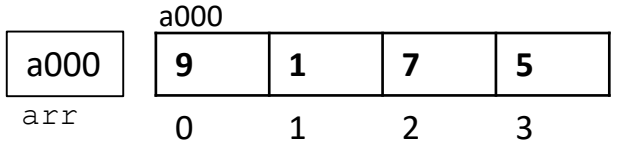
```
lastP->next=newP.
```

d. **Check that the data is good for the next iteration**: Before iteration for index 3 (when j=3) is my data as expected? No because `lastP` is still abcd, but now the last node is at address dabc => update `lastP` as `lastP=newP;`

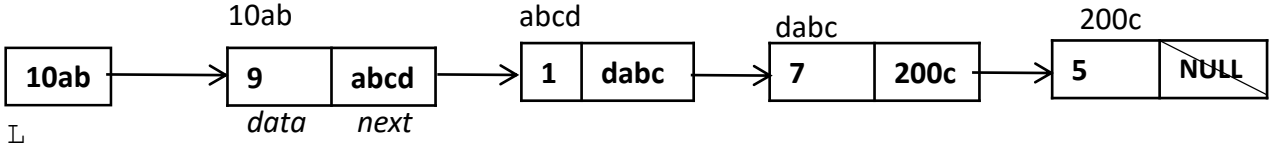


Step 3 - Creating a linked list with data from an array of int

Given data:
array of int, `arr` and int N Drawing:



Data to be created:
Single linked list. Drawing:



Solve the problem using the relation between data and loop iteration continued.

Step 3. **solved in the end**- **Check the state at the end of the loop. Will the problem be solved?**
After the iteration for the last index (j=3), do we have the entire list? – yes, it seems to be so, and the last node points to NULL (indicate the end of the list) thanks to our choice for `newP->next = NULL`

Step 4 - Creating a linked list with data from an array of int

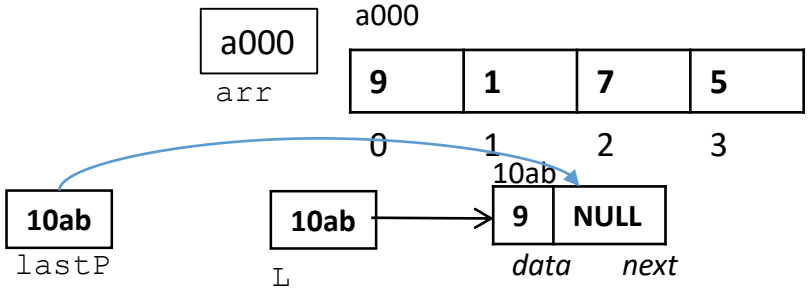
Solve the problem using the relation between data and loop iteration continued.

Step 4. **Check the FIRST iteration of the loop**. What data will it use? Will this be a special case?
Yes. It is a special case because:
a. The address of the first node must be copied in box L (the others are not written in L)
b. When creating the first node (for the number 9), there is NO other node in the list, thus there is no `lastP`, thus the code in the loop may break.

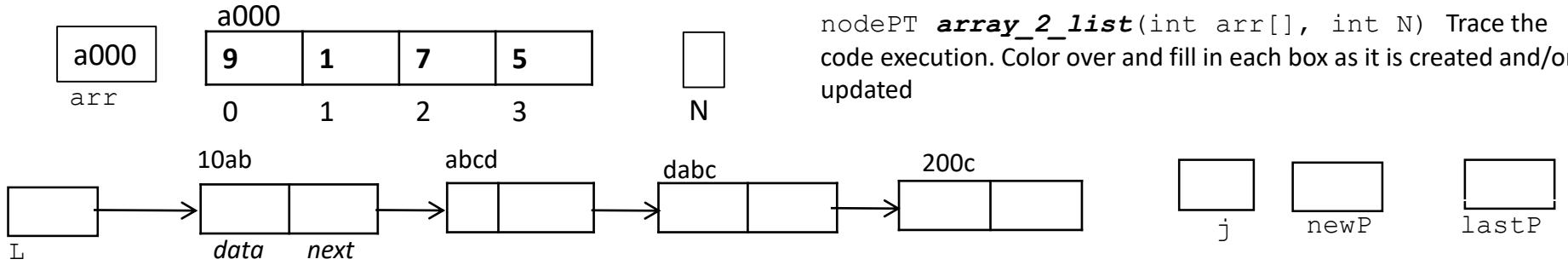
⇒ **Treat this special case separate, NOT in the loop.** This is **just my personal choice** => modify the loop to start at index 1 , not 0 (`for(j=1;j<N;j++)`) (the other way is to create a special case inside the loop for j==0. There are pros and cons to both options).

⇒ Create a node, store its memory address in L :
`nodePT L = malloc(sizeof(struct node))`.
Write data in it: `L->data = arr[0]; L->data = NULL` . How will this node be used by the loop? It is currently the last node in the list, thus make `lastP` point to it by copying its address in `lastP` (`lastP = L`). Check what is expected of `lastP`? It is expected that it point to NULL (i.e. next is NULL.) . It does!

Program state:



Note that even if the array had size 1, the list will be correct (last node points to NULL) (This is one possible special case.)

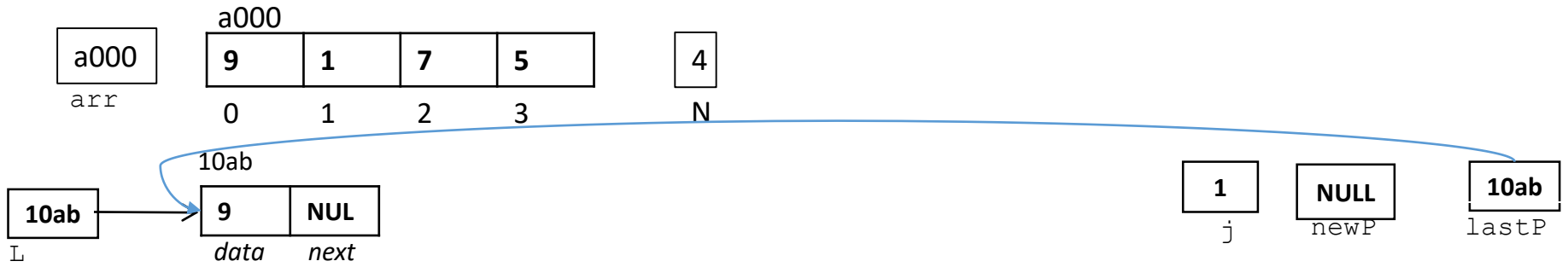


nodePT ***array_2_list***(int arr[], int N) Trace the code execution. Color over and fill in each box as it is created and/or updated

```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)       {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Note: this code can be written even simpler, but this version is very explicit and can be applied to other scenarios.



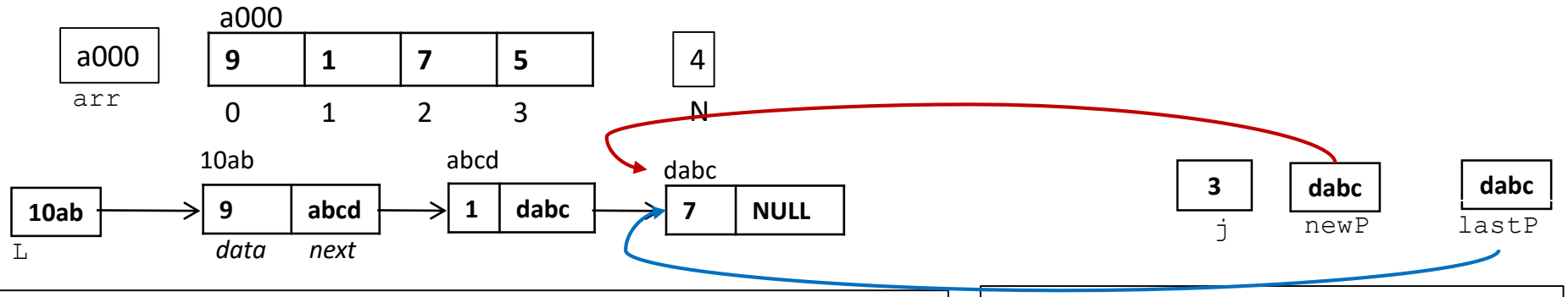

```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)   {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

PROGRAM STATE (all data - all memory used and its content)

Above is the program state just before iteration for j=1 starts (immediately after executing j=1)

This is in text form (to be used in online exam)
arr=(a000; 9,1,7,5); N=(…; 4)
L=(…;10ab),
newP=(…;NULL), lastP=(…;10ab)
j=(…;1)
(10ab;9,NULL)

```
Assume:
typedef struct node * nodePT;
struct node {
     int data;
     struct node * next;
};
```




```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)   {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

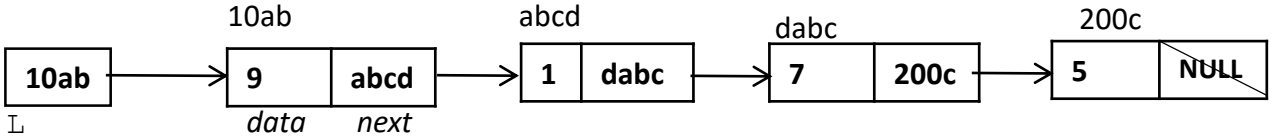
PROGRAM STATE (all program data) just before iteration for j=3 starts (immediately after j is updated to 3). All the data shown is consistent with what the program does (even the NULL in dabc.

This is in text form (to be used in online exam)
arr=(a000; 9,1,7,5); N=(...; 4)
L=(...;10ab),
newP=(...;dabc), lastP=(...;dabc)
j=(...;3)
(10ab;9,abcd)->(abcd; dabc)->(dabc; NULL)

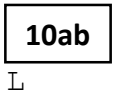
```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Delete an entire list

Given data:



Final data:

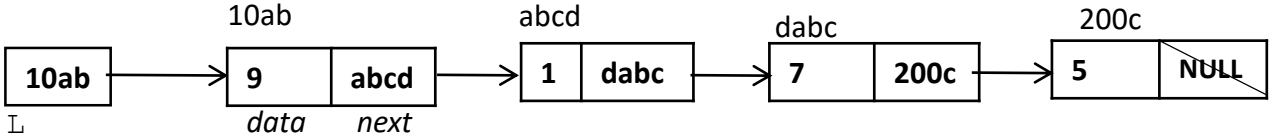


- Function signature: `void delete(nodePT L);`
- STEPS.
  - **1: loop** - decide roughly what the loop does (overall and in one iteration)
  - **2a: identify property** - What is the expected **program state before iteration j**. (CLEARLY state what each variable holds: each variable must have a clear meaning and must hold specific data (related to processing the first (j-1) data.
  - **2b: j -> (j+1)** - **assume** the property holds before iteration j and prove/check it holds before iteration (j+1).

    After the current iteration, j, the variables will hold the same information but related to processing the first j data.
  - **3: solved in the end**- check that when the loop finished, the problem is solved
  - **4: fix start** - check and fix so that the data has the property right before the FIRST iteration starts. Most times, this needs fixing.

Delete an entire list

Given data:



Final data:

- Function signature: `void delete(nodePT L);`
- STEPS.
  - **1: loop** - decide roughly what the loop does (overall and in one iteration)

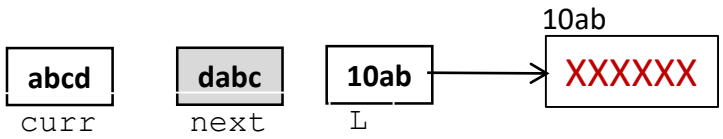
    Iterates over every node, and in one iteration it deletes one node. Use name `curr`

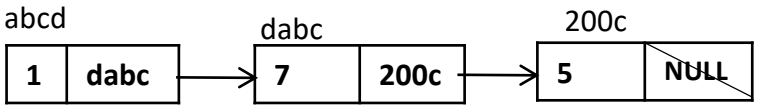
    ```
    for every node, curr
        work needed to delete and free curr
    ```
  - **2a: identify property** - What is the expected **program state before iteration j**. (CLEARLY state what each variable holds: each variable must have a clear meaning and must hold specific data (related to processing the first (j-1) data.

    The first (j-1) nodes are deleted. Nodes before `curr` are deleted and freed
  - **2b: j -> (j+1)** - **assume** the property holds before iteration j and prove/check it holds before iteration (j+1)

    If free(curr), we lose the link to the next node (we do not know the number dabc) => need another pointer variable to hold it => use name `next` . Order matters!!!!

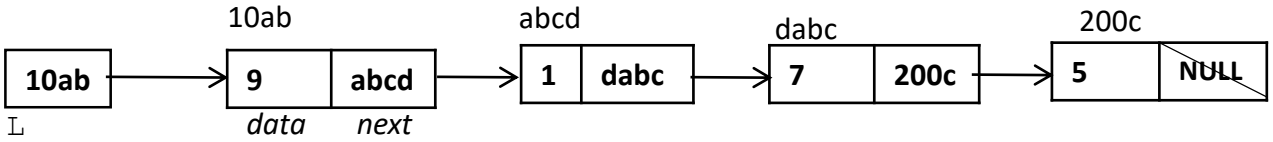
```
next = curr->next;   // line 21
free(curr);          // line 22
curr = next;         // line 23
```





Delete an entire list

Given data:



Final data:

- Function signature: `void delete(nodePT L);`
- STEPS.
  - **3 : solved in the end** - check that when the loop finished, the problem is solved

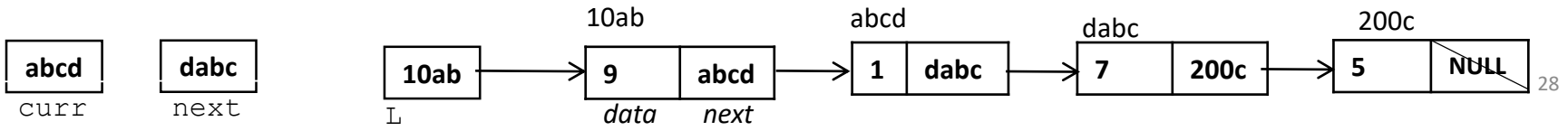
    In the last iteration `curr` points to the last node, 200c, and it deletes in that iteration. L should be fixed to NULL after the loop: `L=NULL;`
  - **4: fix start** - check and fix so that the data has the property right before the FIRST iteration starts. Most times, this needs fixing.

    The only variables used are `curr` and `next`.

    `curr` should point to the first node: `curr=L`

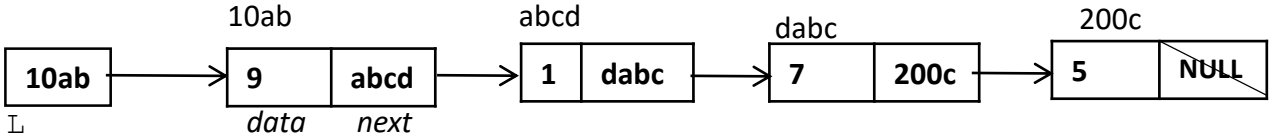
    The CONTENT of after is set before it is used, so it is ok.

```
curr = L;
L=NULL; // if at the end of a function, this can be skipped
while(curr!=NULL){
      next = curr->next;   // line 21
      free(curr);          // line 22
      curr = next;         // line 23
}
```



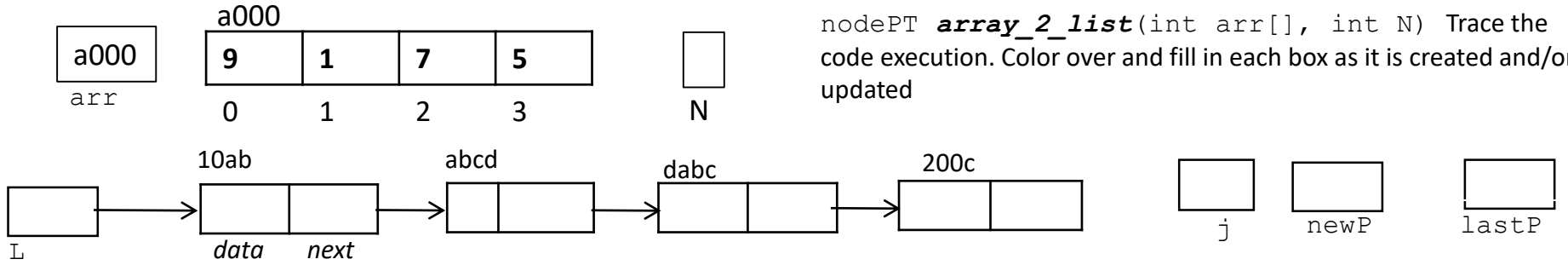
Delete an entire list

Given data:



Final data:


```
// Time complexity: Θ(N),
// where N is the size of the list
nodePT destroy_list(nodePT L) {
    nodePT next, curr;
    curr = L;
    while (curr!=NULL) {
        next = curr->next;
        free(curr);
        curr = next;
    }
    return NULL; // to update pointer in caller code
}
```



nodePT ***array_2_list***(int arr[], int N) Trace the code execution. Color over and fill in each box as it is created and/or updated

```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)       {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Note: this code can be written even simpler, but this version is very explicit and can be applied to other scenarios.

nodePT ***array_2_list***(int arr[], int N)
Trace the code execution. Draw the data.

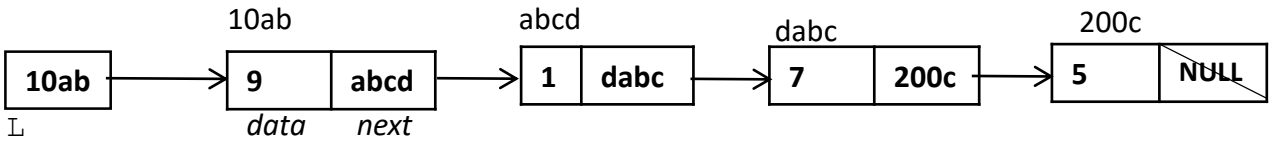
```
// creates a single linked list from an array
nodePT array_2_list(int arr[], int N)  {
  int j;
  nodePT lastP = NULL, newP=NULL;
  nodePT L = malloc(sizeof(struct node));
  L->data = arr[0];
  L->next = NULL;
  lastP = L;
  for (j = 1; j<N; j++)       {
     newP = malloc(sizeof(struct node));
     newP->data = arr[j];
     newP->next = NULL;
     lastP->next = newP;
     lastP = newP;
  }
  return L;
}
```

```
Assume:
typedef struct node * nodePT;
struct node {
    int data;
    struct node * next;
};
```

Note: this code can be written even simpler, but this version is very explicit and can be applied to other scenarios.

Delete an entire list, L: `nodePT destroy_list(nodePT L)`

Given data:

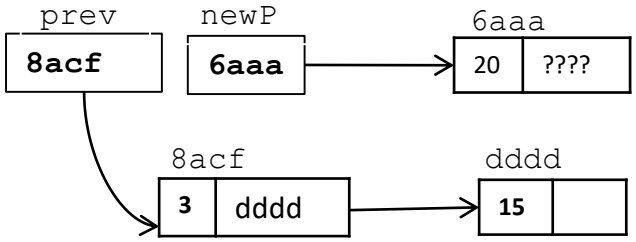


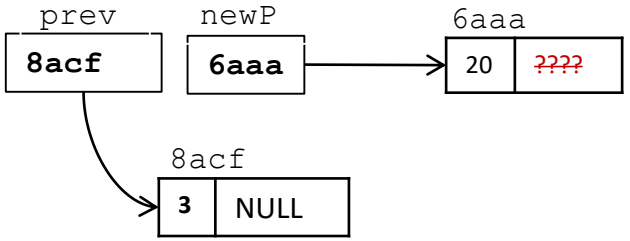
Final data:

Insert a node after a given node – node operation

```
/* Inserts newP after the node "prev".
Note that this is works on nodes. It does not matter how a list is
represented. prev is just a node.   */
void insert_node_after(nodePT prev, nodePT newP) {
    if (((prev == NULL)||(newP==NULL)) {
        printf("\n Cannot insert after a NULL node. No action
taken.");
    } else {
        newP->next = prev->next; //5
        prev->next = newP;  //6
    }
}
```

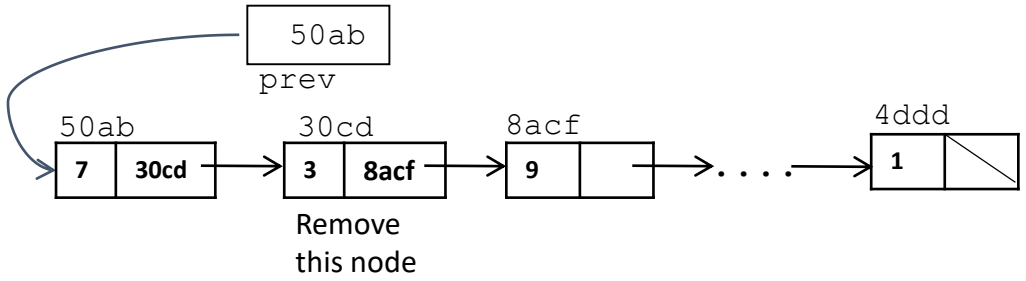
Does this code work well when prev->next is NULL ?





Delete a node after a given node – node operation

```
/* Delete the node after the node "prev".
   Note that this is works on nodes. It does not matter how a list is
represented. prev is just a node.*/
void delete_node_after(nodePT prev) {
    if (prev == NULL) {
        printf("\n Cannot delete after a NULL node. No action taken.");

    } else {
        nodePT toDel = prev->next;  // 3
        if (toDel != NULL){         // 4
            prev->next = toDel->next; // 5 this crashes if toDel is NULL
            free(toDel);          // 6
        }
    }
}
```



Swap the next 2 nodes after node prev

HINT: ***When swapping, NAME the nodes*** to avoid overwriting a link. Below lines 8,9,10 can be executed in any order. If not named, a specific order would be needed.

```
// Swaps 2 nodes after prev. If prev is NULL or not enough nodes, it does nothing.
void swap_2_after(nodePT prev){
```