Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

I2204 - Imperative Programming

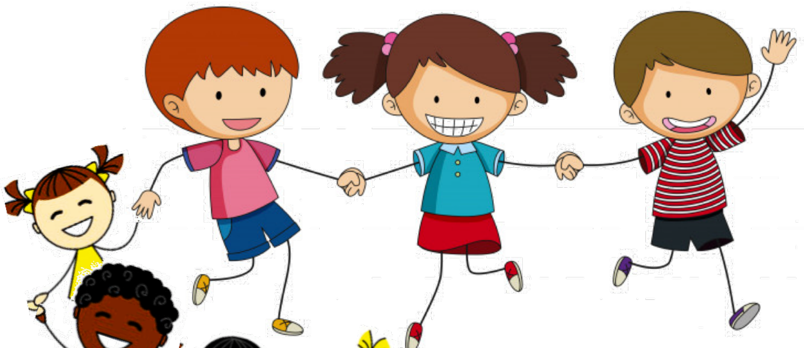
Dr Siba Haidar

Lebanese University
Faculty of Science
BS Computer Science
2nd Year – S3

Linked Lists

Chapter 4

Linked Lists



1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. Linked Lists Variants

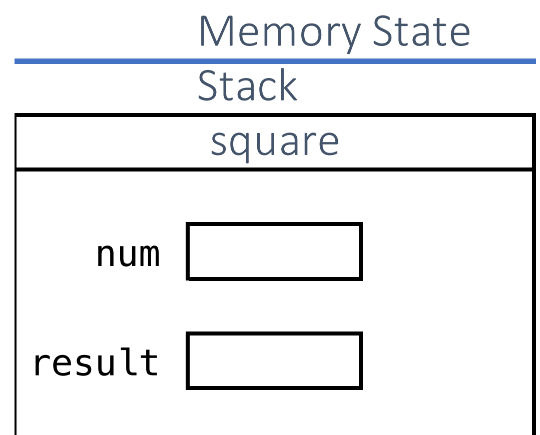
Linked Lists

1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. Linked Lists Variants

The Local Memory (The Stack)

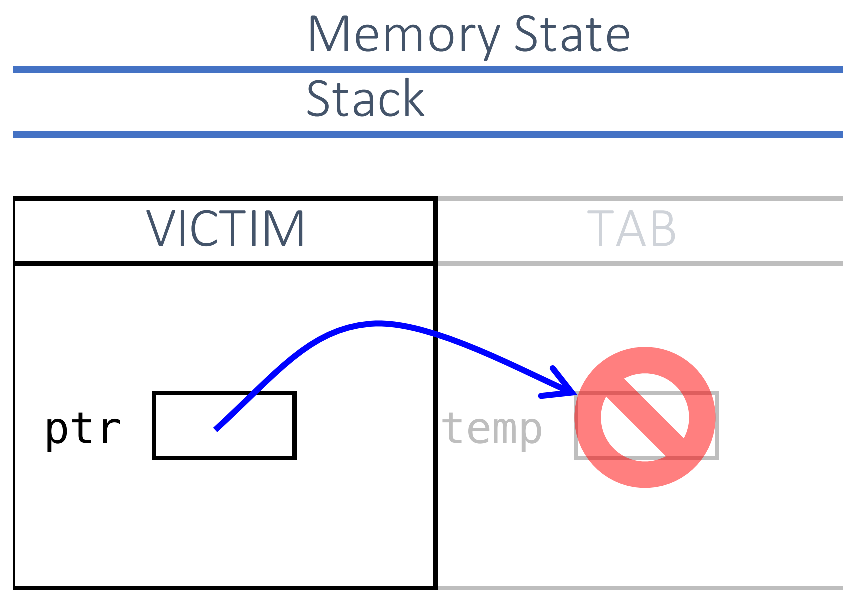
- most common use
- behaves as a first-in-last-out buffer
- essential element: Stack Pointer register
- variables allocated on the stack are
  - stored directly
  - fast access
  - "locals": lifetime tied to the function where they are declared
    - function runs → allocated
    - function exits → deallocated

```
int square(int num) {
   int result;
   result = num * num;
   return result;
}
```



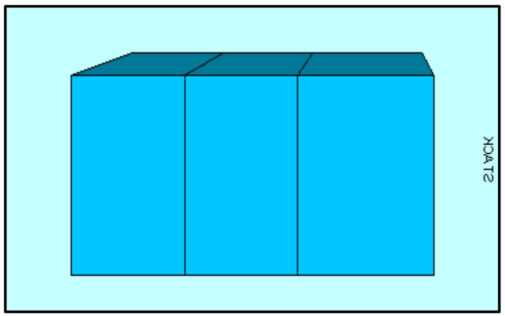
The Ampersand (&) Bug — TAB

```
int* TAB() {
   int temp;
   return(&temp);
}

void Victim() {
   int* ptr;
   ptr = TAB();
   *ptr = 42;
}
```



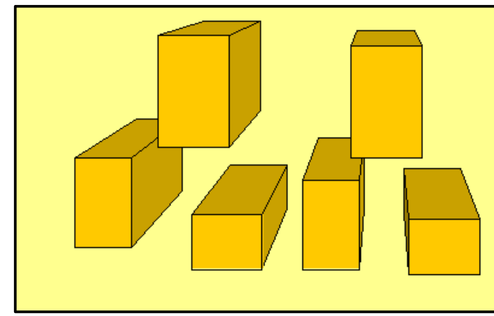
Memory

Local: The Stack



- used for static memory allocation
- **automatic**
  - variables allocated | deallocated automatically on function call | exit

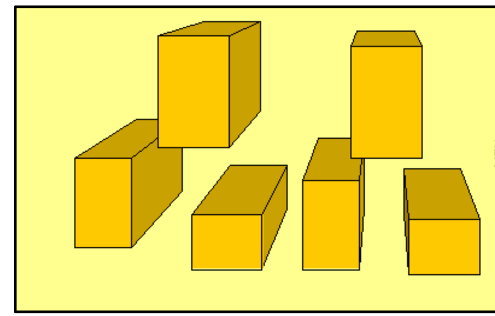
Dynamic: The Heap



- used for dynamic memory allocation
- **nothing happens automatically**
  - explicit request of allocation | deallocation of memory

Memory

Dynamic: The Heap



Advantages

- lifetime controlled
- size controlled
- greater control of memory

Disadvantages

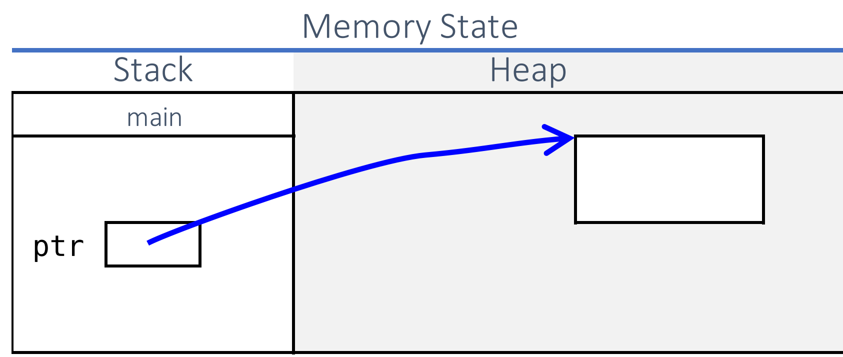
- more work
- more bugs
- greater responsibility

- used for dynamic memory allocation
- **nothing happens automatically**
  - explicit request of allocation | deallocation of memory

Dynamic Allocation Functions

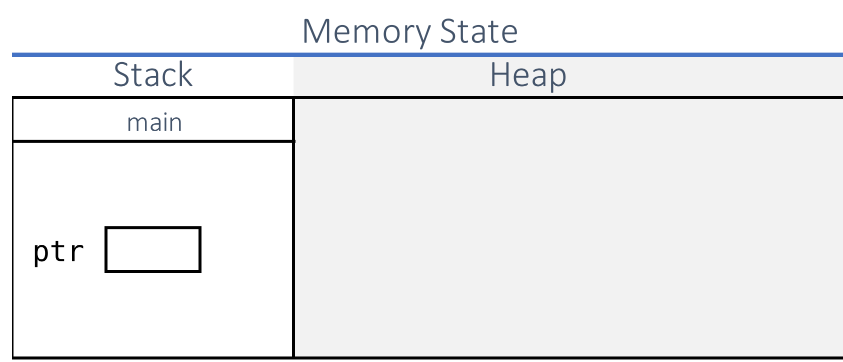
```c
#include <stdlib.h>
```

- core of allocation system consists of functions `malloc()` and `free()`



- `malloc():` allocates memory from portion of remaining free memory

- `free():` releases memory and returns it to system; and so may be re-used to satisfy future allocation requests



Allocation

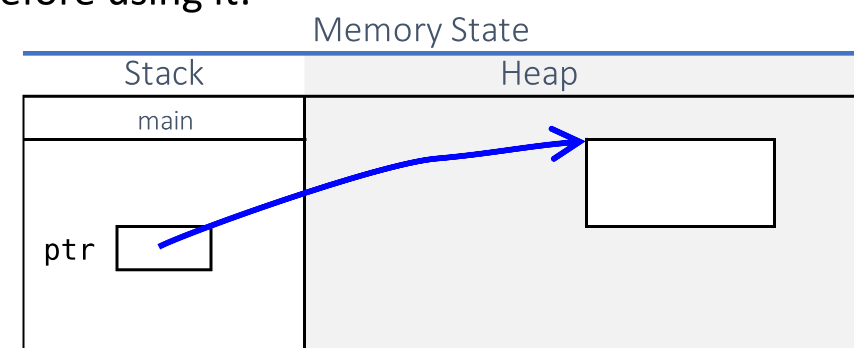
```
void * malloc(size_t number_of_bytes);
```

- program can explicitly request areas, or "blocks", of memory for use by calling function malloc, which, reserves in heap a block of memory of requested size and returns a pointer to it

```
int *ptr;
ptr = (int *) malloc(50 * sizeof(int));
```

- check if the memory was really allocated before using it:

```
if(!ptr){
  printf("Out of memory.\n");
  exit(1);
}
```



Deallocation

```
void free (void *p);
```

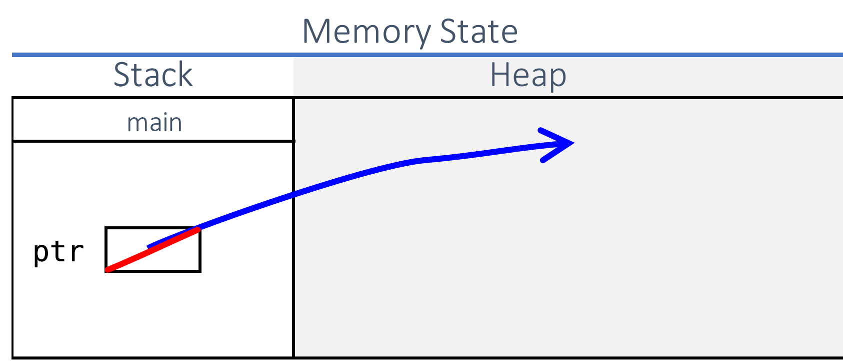
- if program finished using a block of memory, it must make an explicit deallocation request to indicate so to heap manager, which updates its private data structures.

```
free(ptr);
```

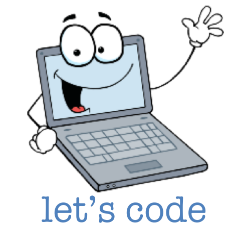
- remember to reset `ptr` after free

```
ptr = NULL;
```

Never call free() with an invalid argument;
it will destroy the free list.

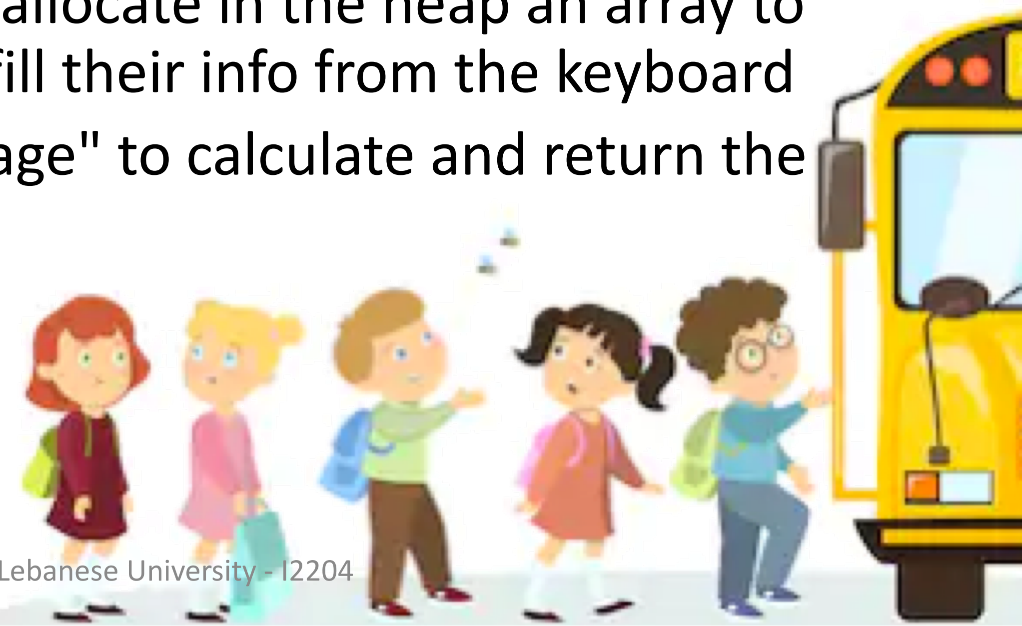


Exercise: Allocate & Fill



Write a program which:

- defines a struct type student (name + 6 marks)
- asks for the exact number of students
- asks a function "allocFill" to allocate in the heap an array to hold the students info, and fill their info from the keyboard
- asks another function "average" to calculate and return the class average
- displays the class average
- frees the dynamic memory



Linked Lists

1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. Linked Lists Variants

Linked Lists

- useful for 2 reasons
  - data structure used in real programs
  - appreciation of time, space, code issues, useful to thinking about any data structures in general

- great way to learn about pointers:
  - problems are a nice combination of algorithms and pointer manipulation

Why Linked Lists?

- similar to arrays: both store collections of data
- different strategies

- arrays strategy:
    1. entire array is allocated as one block of memory
    2. direct access using [] syntax
- linked lists strategy:
    1. memory allocation for each element separately only when necessary
    2. access is more complex

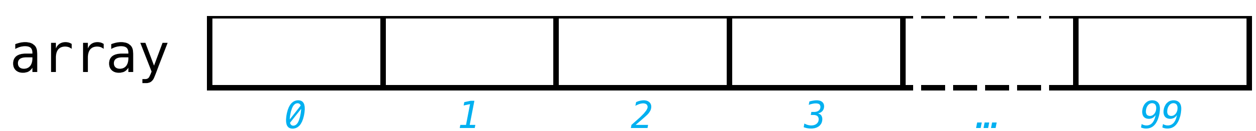
Disadvantages of Arrays

1. fixed size
   – specified at compile time
   – even if deferred until runtime, after that it remains fixed
2. allocate "large enough"
   – most of time 70% of space is wasted
   – if need more, code breaks
   – ++ commercial codes

can allocate array in heap and then dynamically resize it with `realloc()`

– OK but …

3. inserting new elements at the front is expensive
   – because existing elements need to be shifted over to make room



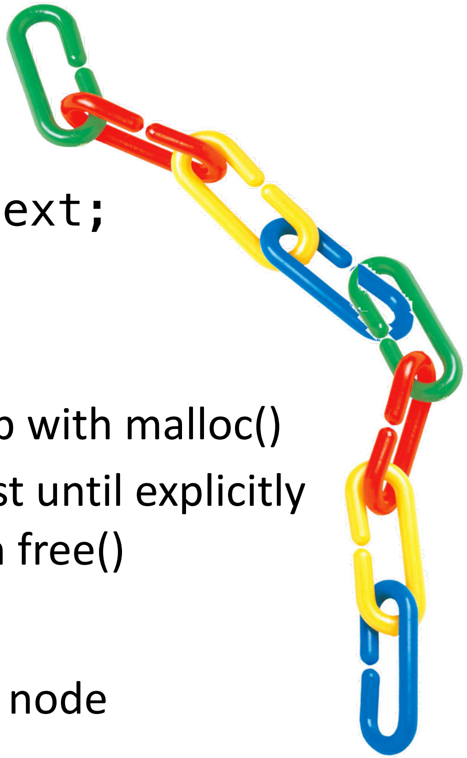
What Linked Lists Look Like

- node: linked list element
- list gets its overall structure by using pointers to connect all its nodes together like links in a chain
- each node contains 2 fields
  - "data" field: store whatever element type list holds for its client
  - "next" field: pointer to link one node to next node

node definition

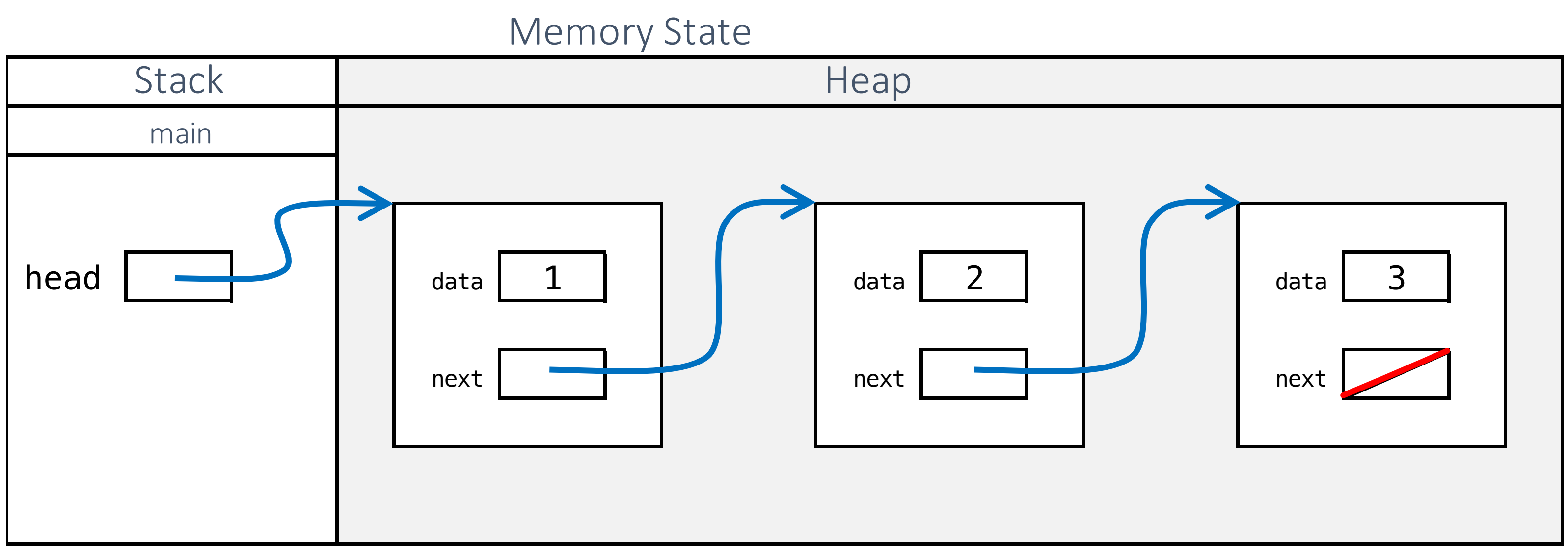
```
struct node {
   int data;
   struct node* next;
};
```

- each node
  - allocated in heap with malloc()
  - continues to exist until explicitly deallocated with free()
- front of list
  - a pointer to first node



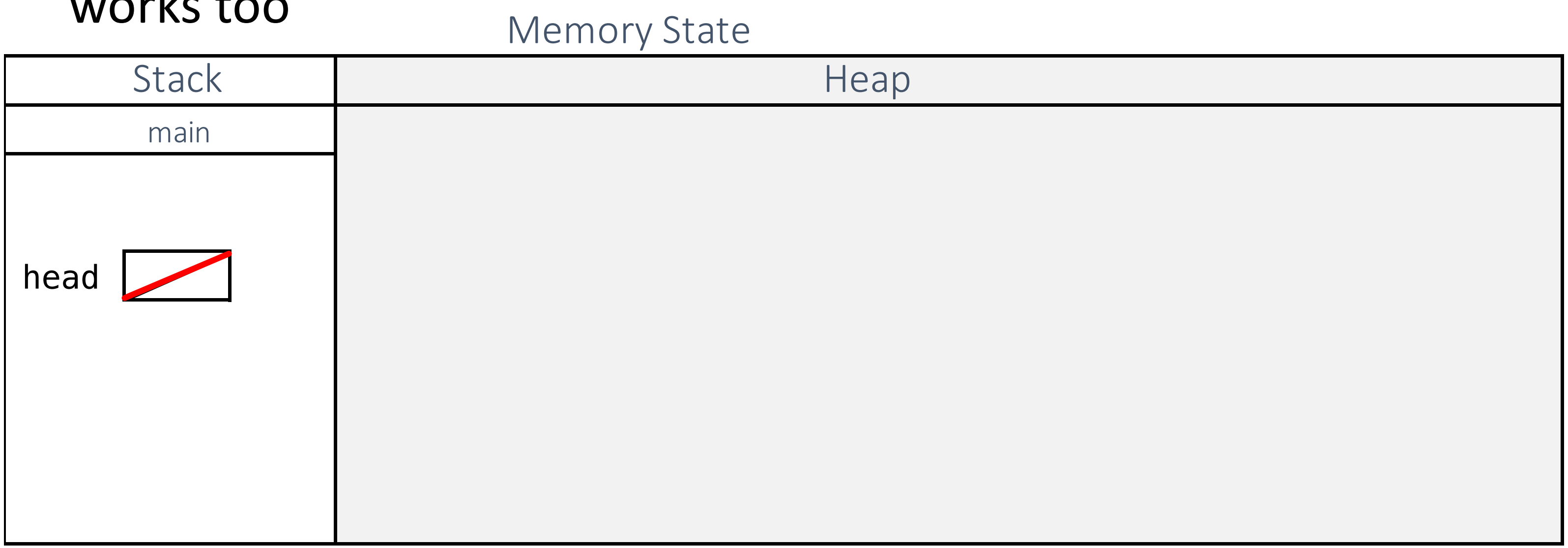
Example : List {1, 2, 3}

- The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.

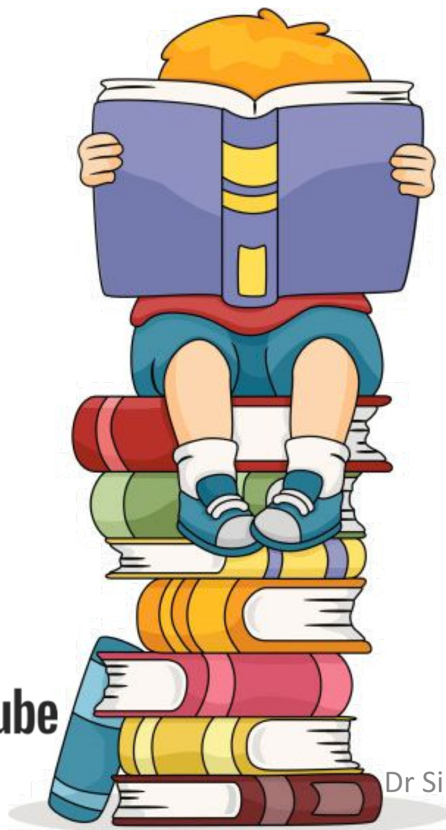


The Empty List — NULL

- empty list — list with zero nodes: NULL head pointer
  - empty list case is "boundary case" for linked list code:
  - good habit to remember to check empty list case to verify that it works too

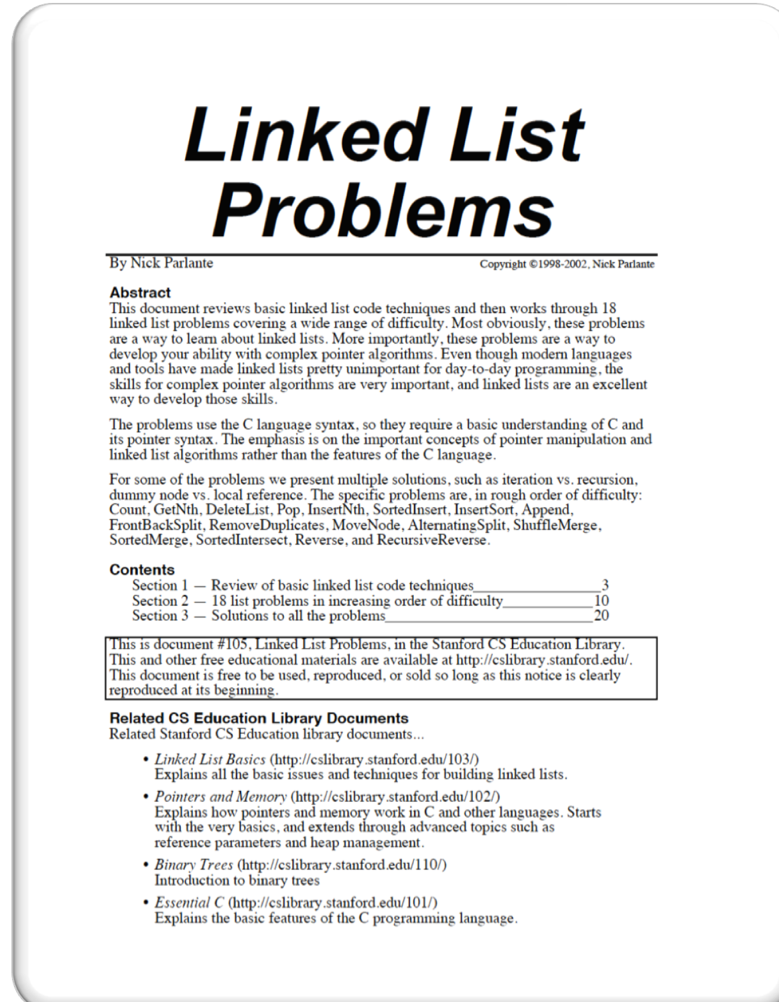


Linked Lists



1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. Linked Lists Variants

YouTube



Linked List Problems

By Nick Parlante

Copyright ©1998-2002, Nick Parlante

Abstract

This document reviews basic linked list code techniques and then works through 18 linked list problems covering a wide range of difficulty. Most obviously, these problems are a way to learn about linked lists. More importantly, these problems are a way to develop your ability with complex pointer algorithms. Even though modern languages and tools have made linked lists pretty unimportant for day-to-day programming, the skills for complex pointer algorithms are very important, and linked lists are an excellent way to develop those skills.

The problems use the C language syntax, so they require a basic understanding of C and its pointer syntax. The emphasis is on the important concepts of pointer manipulation and linked list algorithms rather than the features of the C language.

For some of the problems we present multiple solutions, such as iteration vs. recursion, dummy node vs. local reference. The specific problems are, in rough order of difficulty: Count, GetNth, DeleteList, Pop, InsertNth, SortedInsert, InsertSort, Append, FrontBackSplit, RemoveDuplicates, MoveNode, AlternatingSplit, ShuffleMerge, SortedMerge, SortedIntersect, Reverse, and RecursiveReverse.

Contents

Section 1 — Review of basic linked list code techniques ____ 3
Section 2 — 18 list problems in increasing order of difficulty ____ 10
Section 3 — Solutions to all the problems ____ 20

This is document #105, Linked List Problems, in the Stanford CS Education Library. This and other free educational materials are available at http://cslibrary.stanford.edu/. This document is free to be used, reproduced, or sold so long as this notice is clearly reproduced at its beginning.

Related CS Education Library Documents

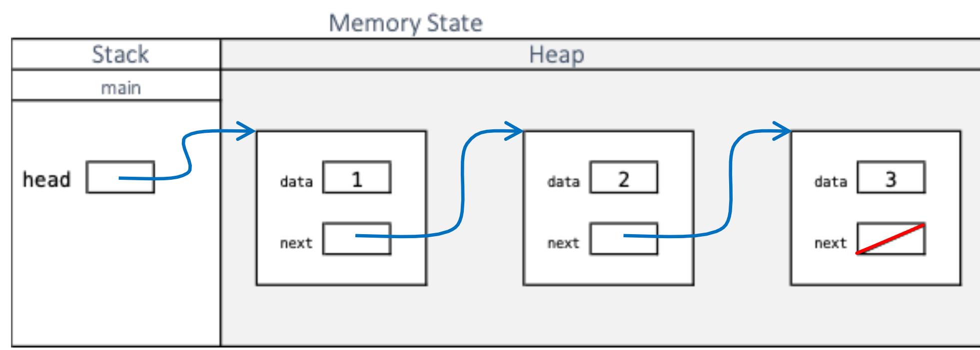
Related Stanford CS Education library documents...

- *Linked List Basics* (http://cslibrary.stanford.edu/103/)
  Explains all the basic issues and techniques for building linked lists.
- *Pointers and Memory* (http://cslibrary.stanford.edu/102/)
  Explains how pointers and memory work in C and other languages. Starts with the very basics, and extends through advanced topics such as reference parameters and heap management.
- *Binary Trees* (http://cslibrary.stanford.edu/110/)
  Introduction to binary trees
- *Essential C* (http://cslibrary.stanford.edu/101/)
  Explains the basic features of the C programming language.

1) Iterate Down a List

- iterate a pointer over all nodes in a list, using a loop
  - copy head pointer into local variable
    - current = head
  - test for end of list with
    - current != NULL
  - advance pointer with
    - current = current->next

```
int length(struct node* head) {
  int count = 0;
  struct node* current = head;
  while (current != NULL) {
    count++;
    current = current->next;
  }
  return count;
}
```



1) Iterate Down a List

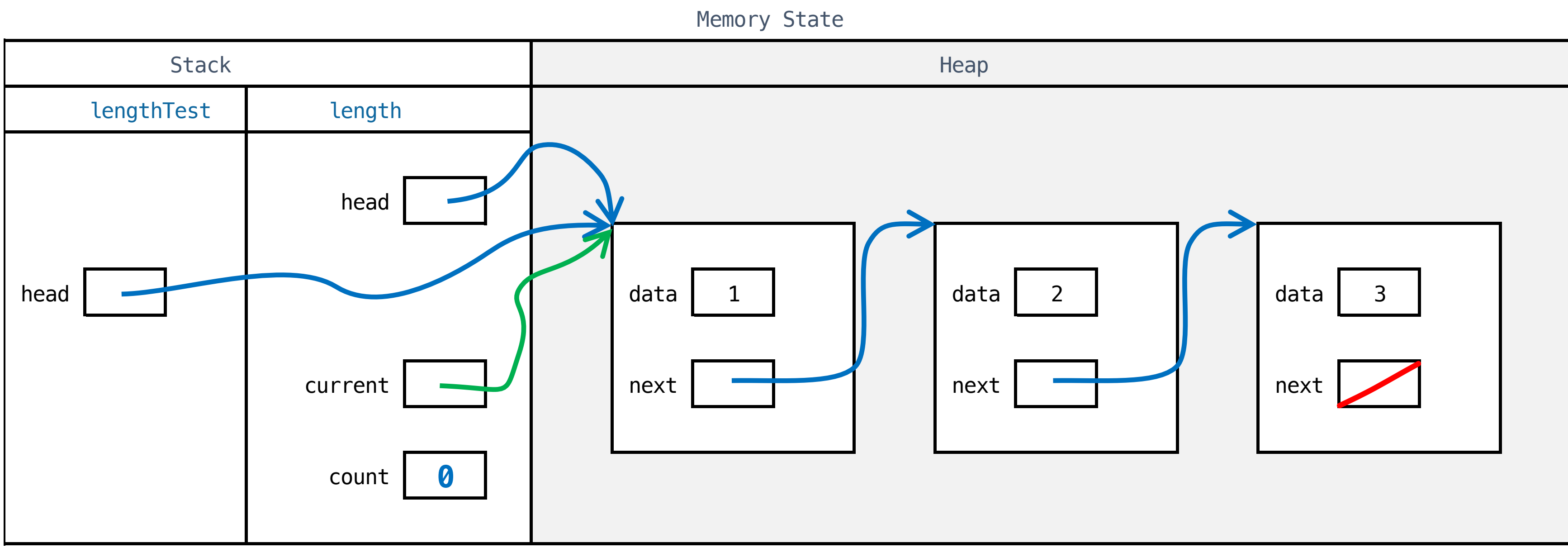
- iterate a pointer over all nodes in a list, using a loop
  - copy head pointer into local variable
    - current
  - test for end of list with
    - current != NULL
  - advance pointer with
    - current = current->next
- for loop makes initialization, test, and pointer advance harder to omit...

```
int length(struct node* head) {
  int count = 0;
  struct node* current = head;
  while (current != NULL) {
    count++;
    current = current->next;
  }
  return(count);
}
```

```
for (current = head;
     current != NULL;
     current = current->next)
  count++;
```

1) Iterate Down a List

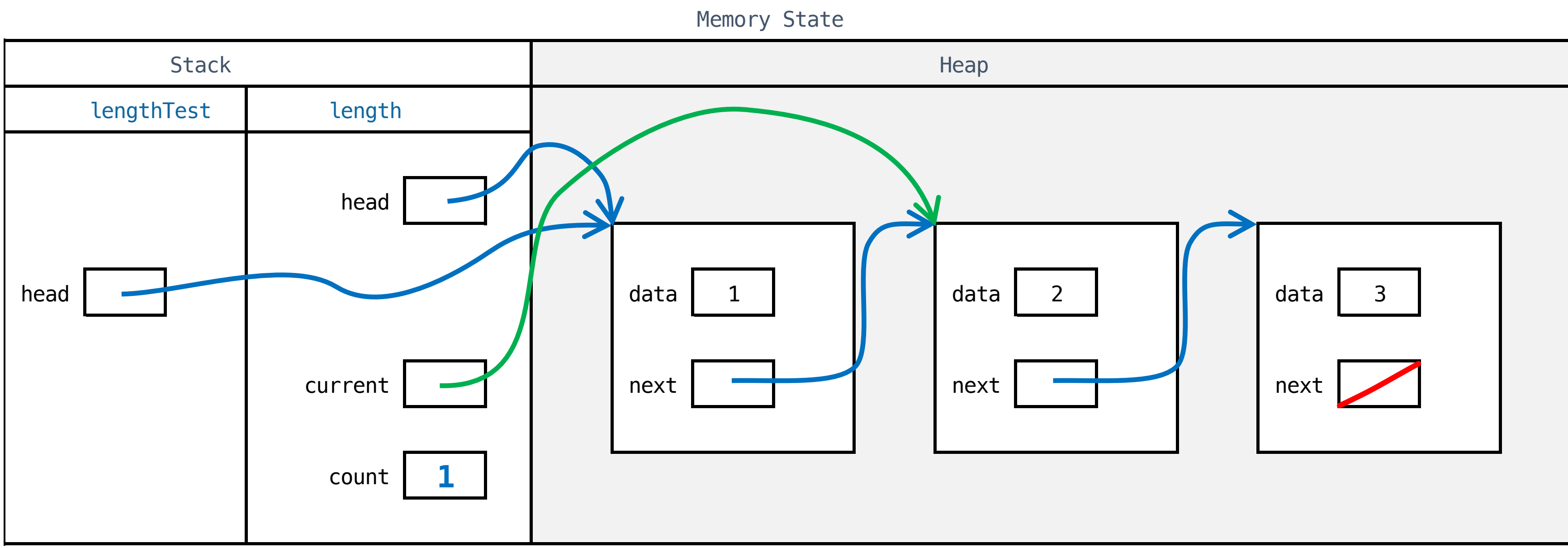
- iteration 1:



memory state

1) Iterate Down a List

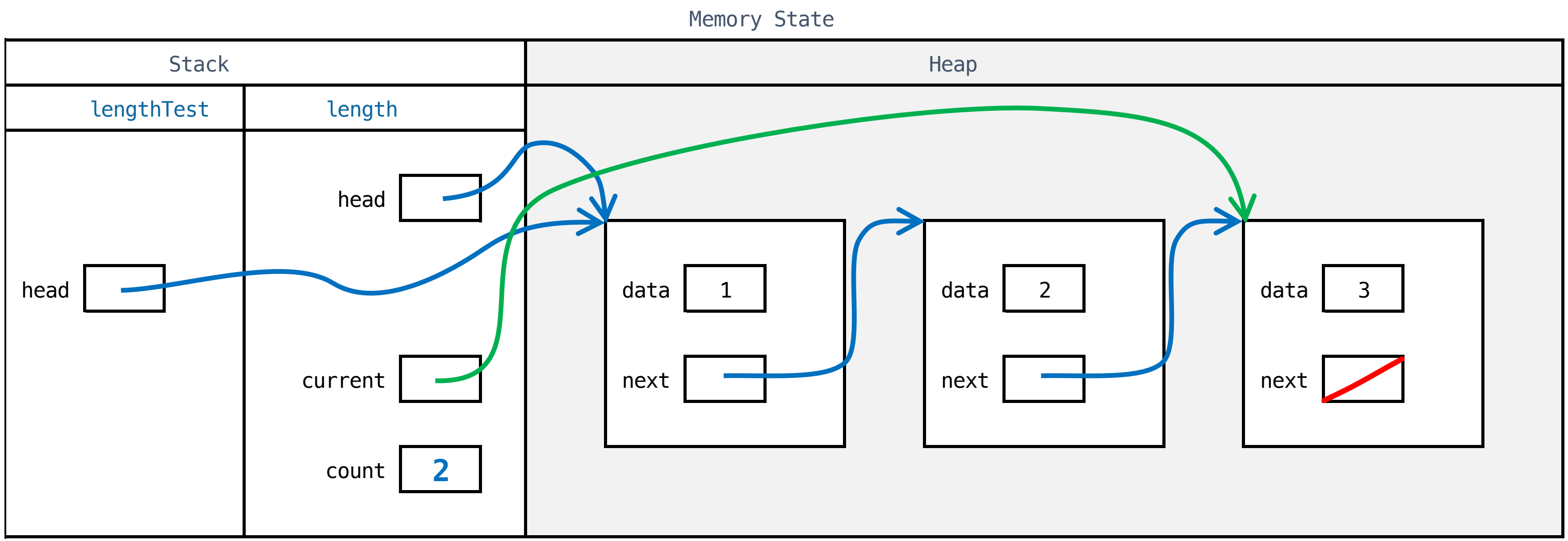
- iteration 2:



memory state

1) Iterate Down a List

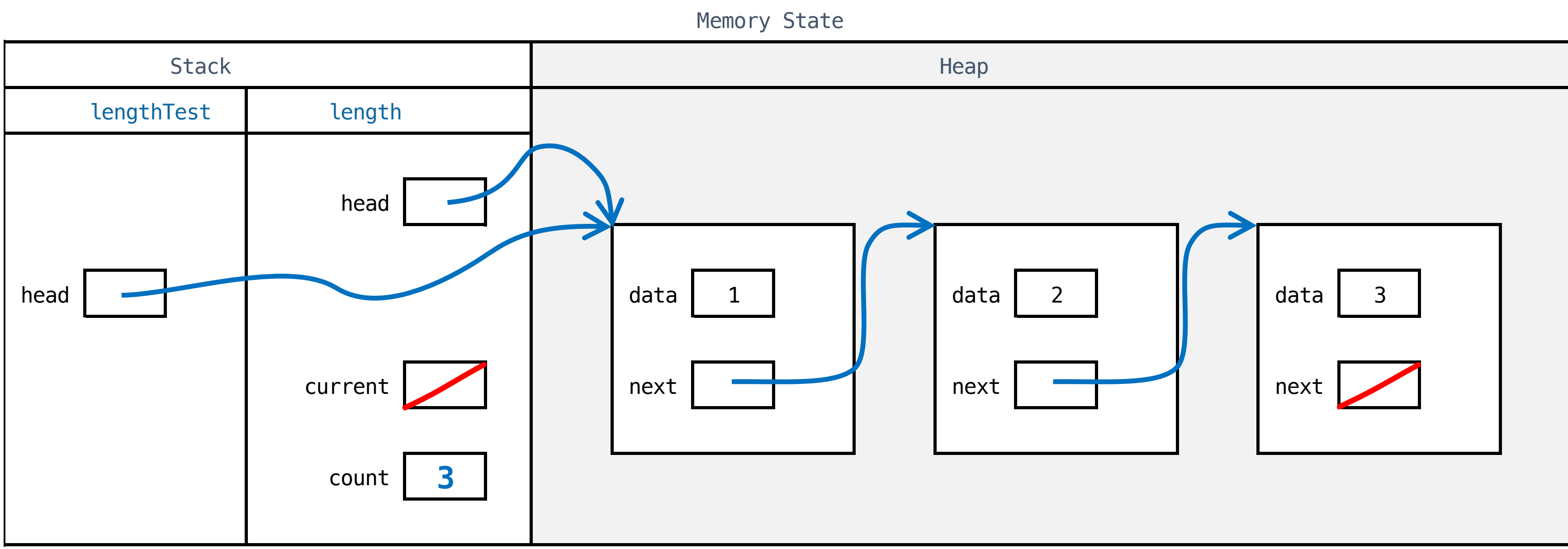
- iteration 3:



memory state

1) Iterate Down a List

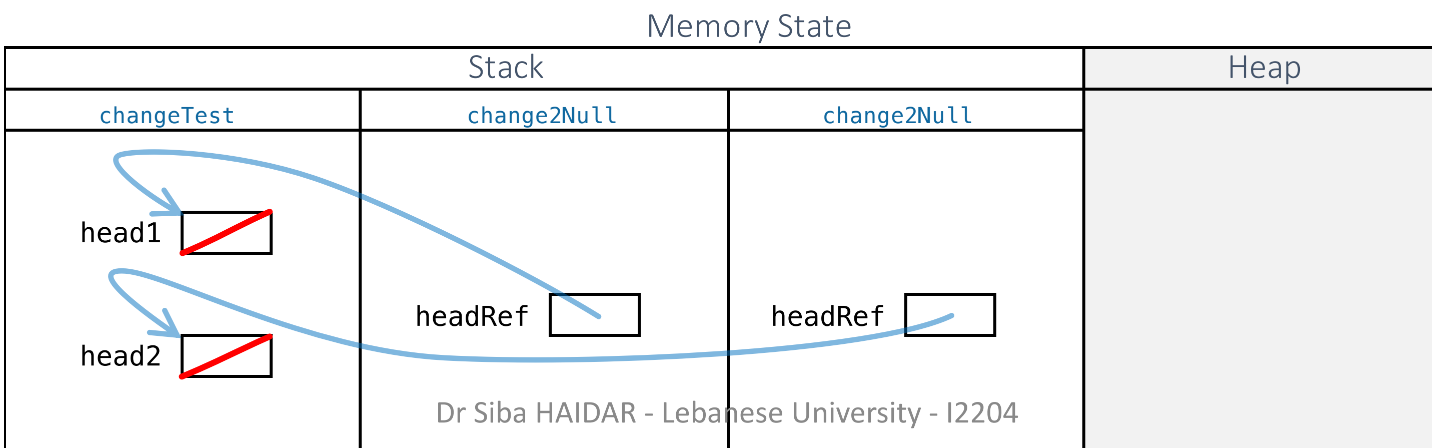
- Stopping condition: `current == NULL`



2) Changing a Pointer With A Reference Pointer

- if functions need to change caller's head pointer → pass *pointer to head pointer*: reference pointer
  - to change a node*, pass a node**
  - use & in call
  - use * in callee function to access and change value

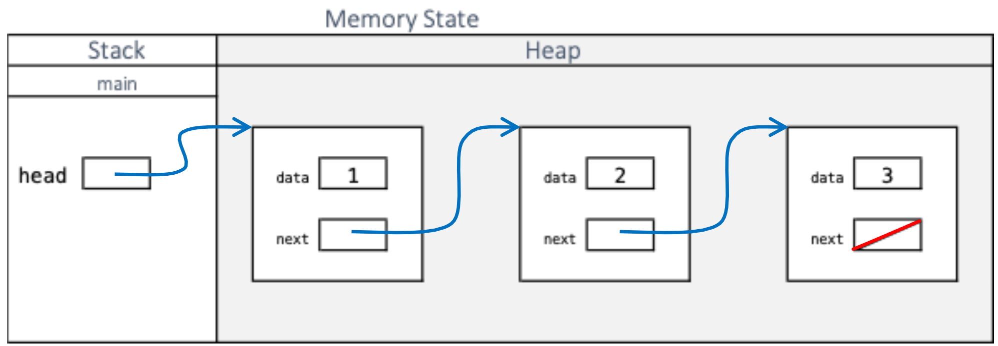
```
void change2Null(node** headRef) {
  *headRef = NULL;
}
void changeTest() {
  node* head1, *head2;
  change2Null(&head1);
  change2Null(&head2);
  printf("%p %p\n", head1, head2);
}
```



Special Application: List Building

- Best Solution
  - independent function that adds a single new node to any list
  - can call function as many times as we want to build up any list
- Classic 3-Step Link In Operation
  - adds a single node to the front of a linked list
  - 3 steps = allocate & fill + link next + link head

→ **Push**: add a node to the head of the list



Push: add a node to the head of the list

- 3-Step Link In operation

1. allocate & fill: allocate the new node in the heap and set its .data to whatever needs to be stored
2. link next: set the .next pointer of the new node to point to the current first node of the list
3. link head: change the head pointer to point to the new node, so it is now the first node in the list

```
void Push (node ** headRef, int d){

  // 1 - allocate & fill
  node * newNode = (node *) malloc (sizeof (node));
  newNode->data = d;

  // 2 - link next
  newNode->next = *headRef;

  // 3 - link head
  *headRef = newNode;

}
```

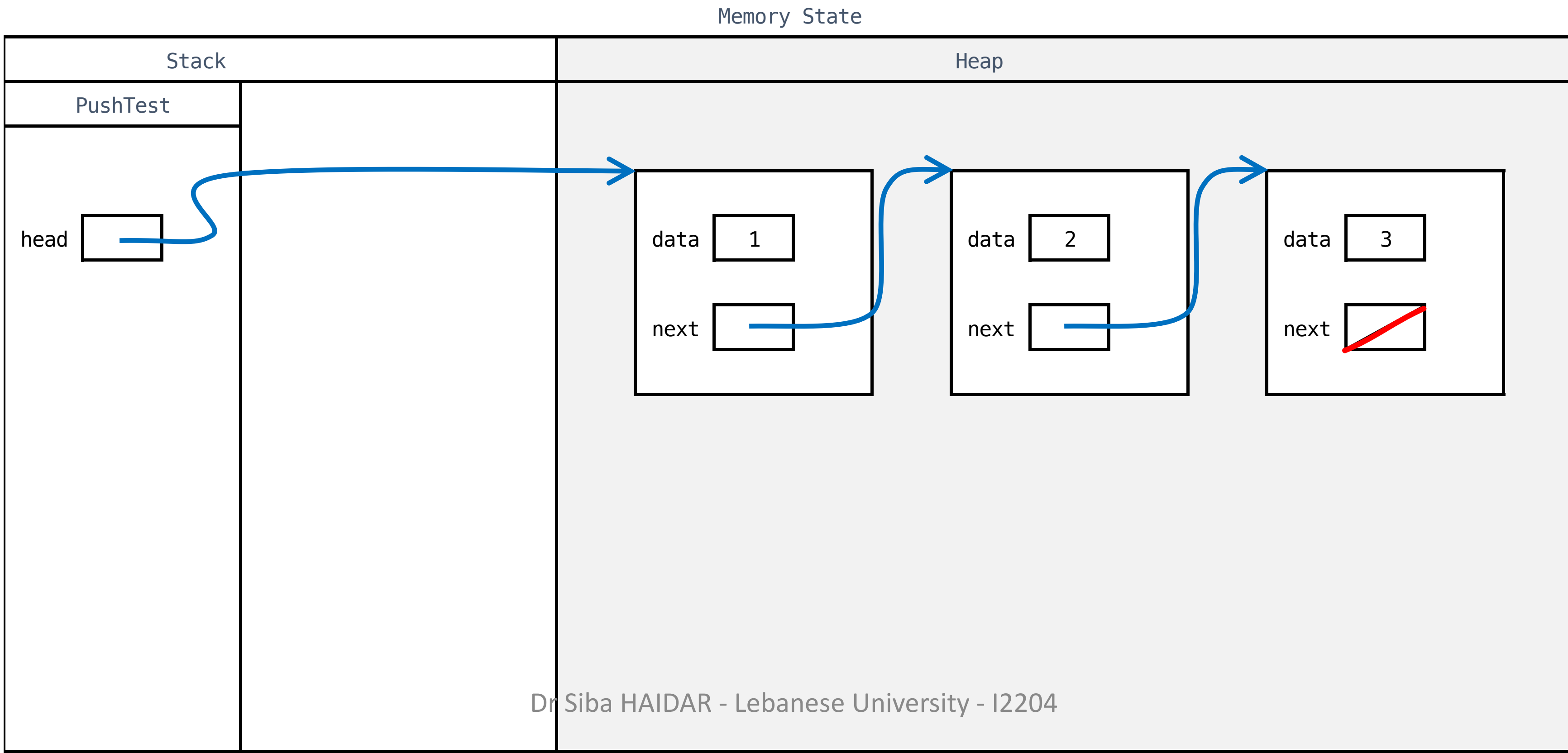
Push Animation

- Suppose we have the list {1,2,3} and we want to push 0 to the head of the list, so it becomes {0,1,2,3}.

```
void Push (node ** headRef, int d){

    // 1 - allocate & fill
    node * newNode = (node *) malloc (sizeof (node));
    newNode->data = d;

    // 2 - link next
    newNode->next = *headRef;


    // 3 - link head
    *headRef = newNode;


}
```

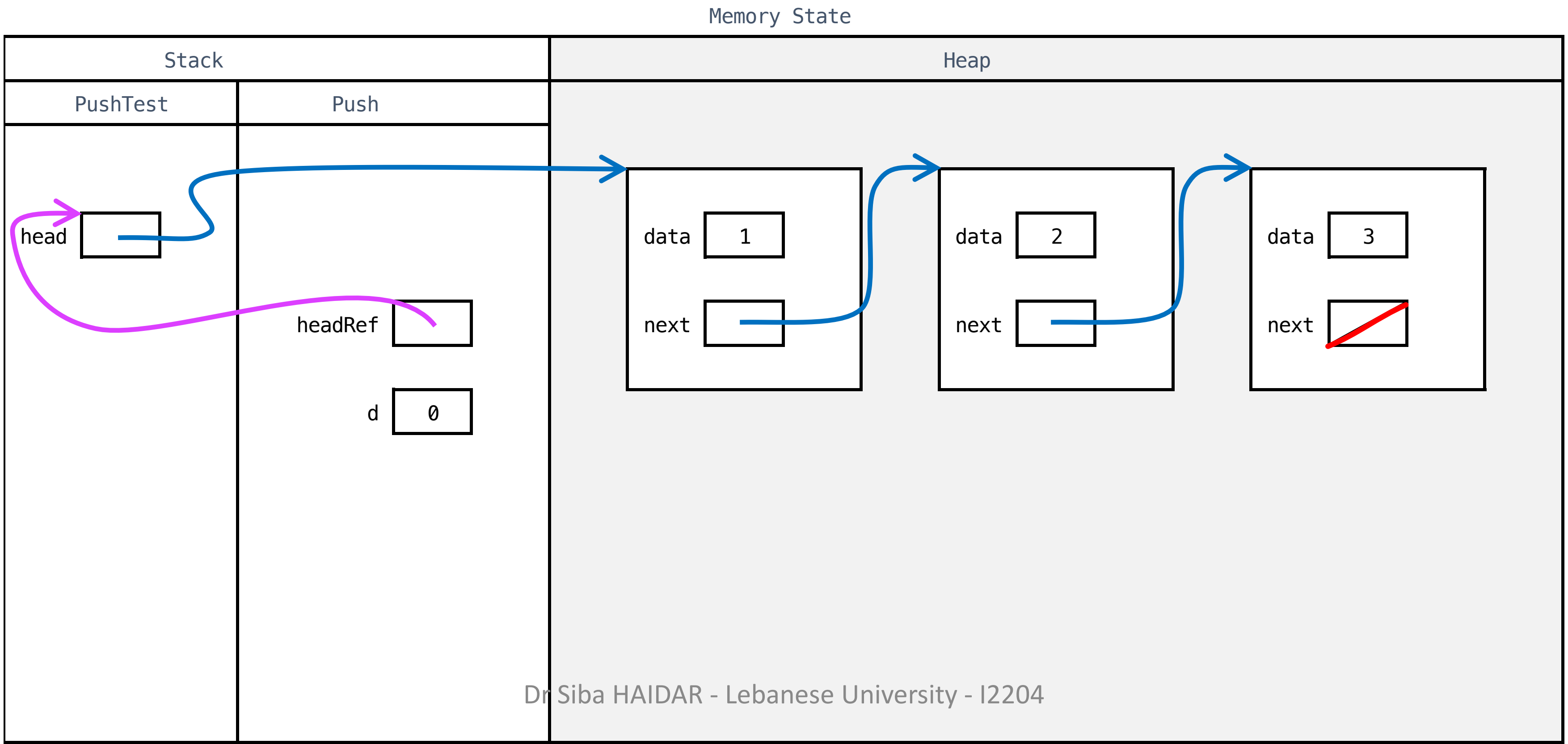
Push Animation (1)

- Initial state: `{1,2,3}`



Push Animation (2)

- Call `Push (&head, 0)` :

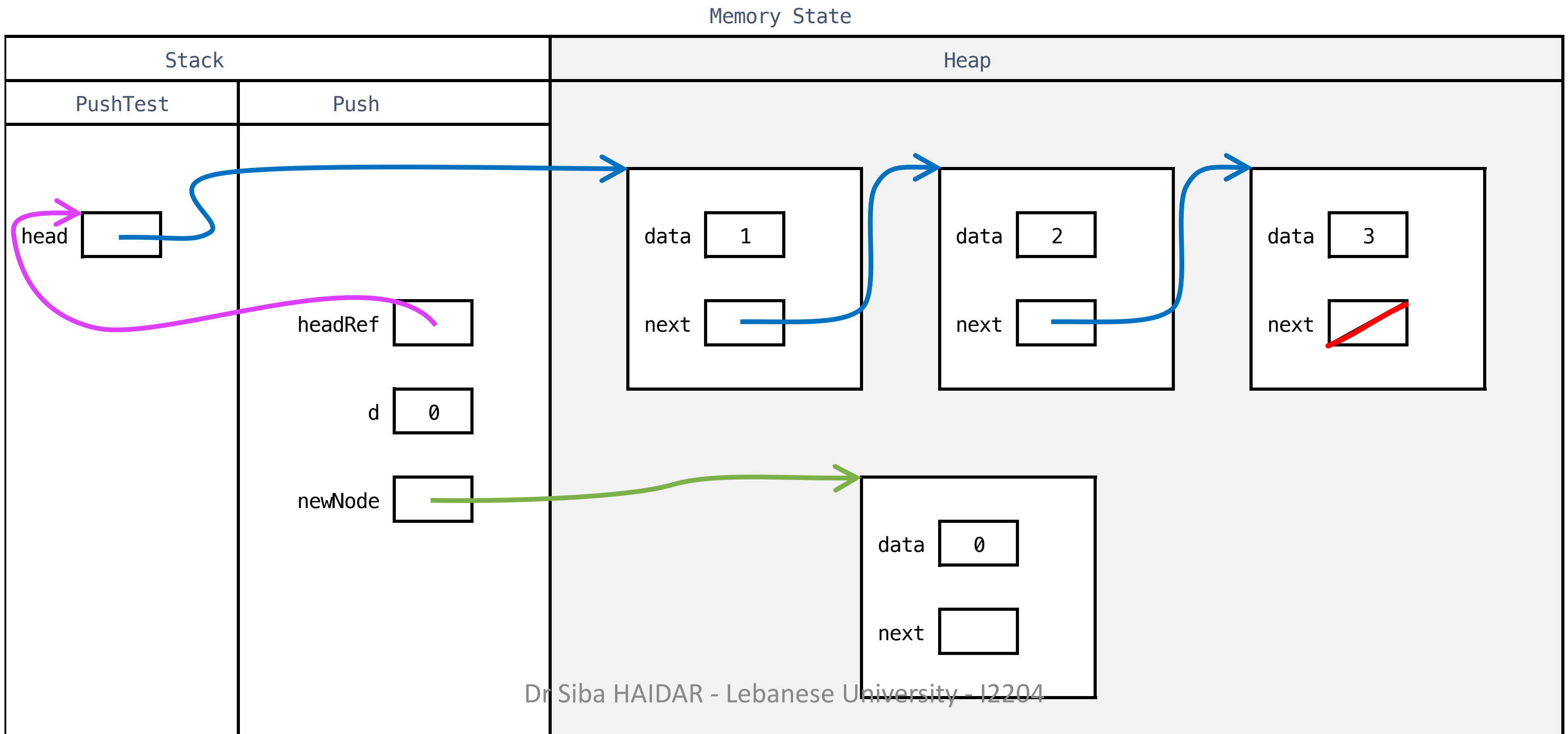


memory state

Push Animation (3)

- 1 – allocate & fill:

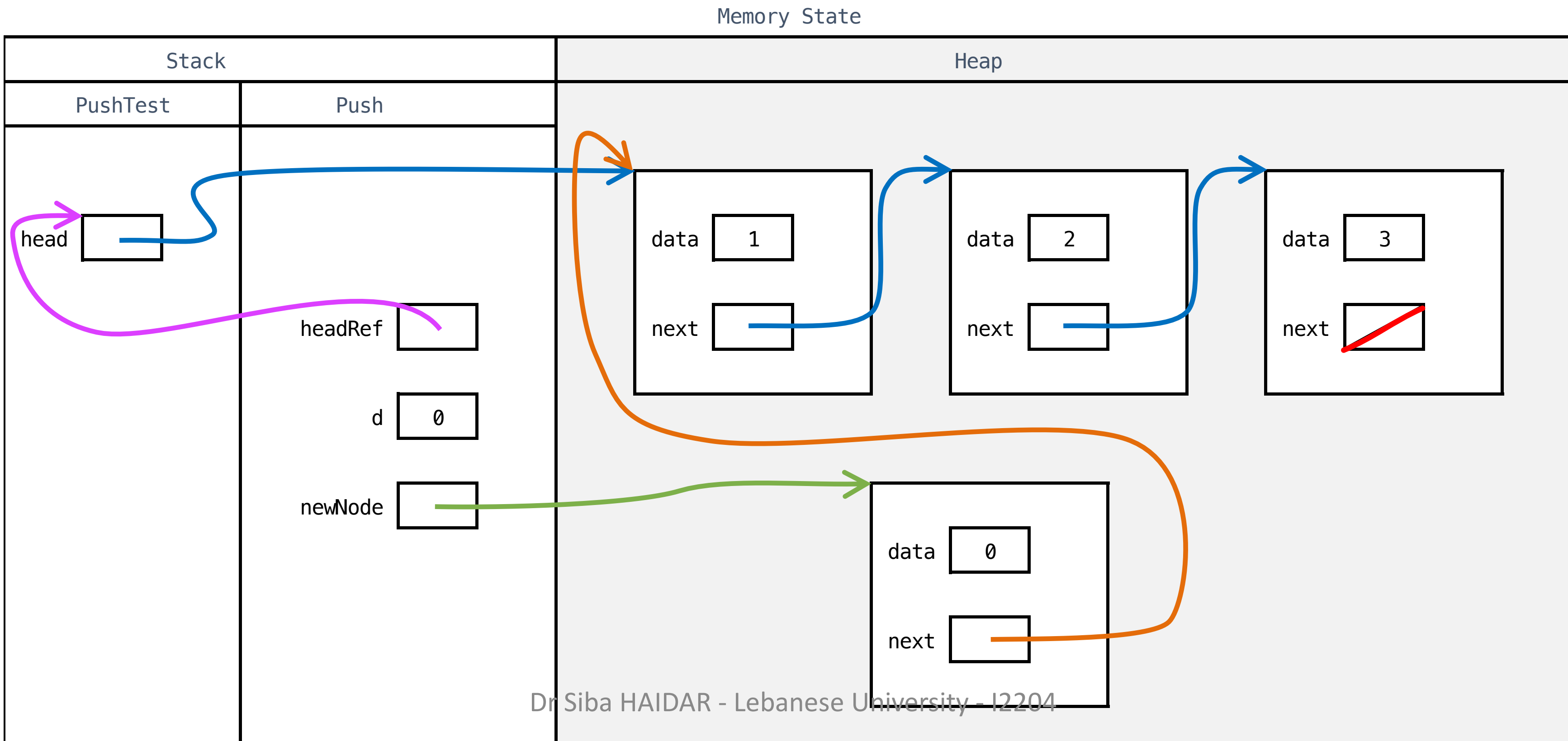
```
node* newNode=(node*)malloc(sizeof(node));
newNode->data = d;
```



memory state

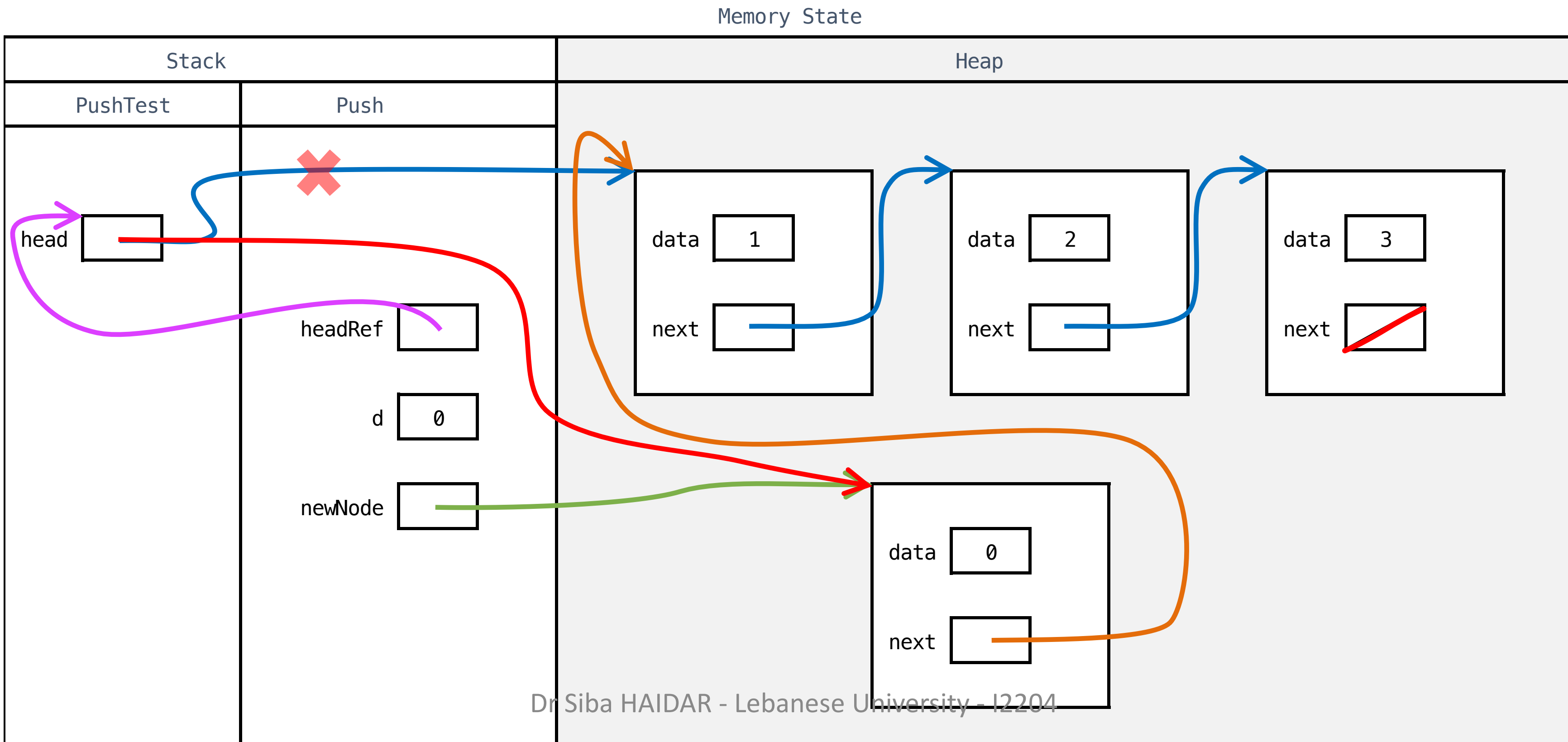
Push Animation (4)

- 2 – link next: newNode->next = *headRef;



Push Animation (5)

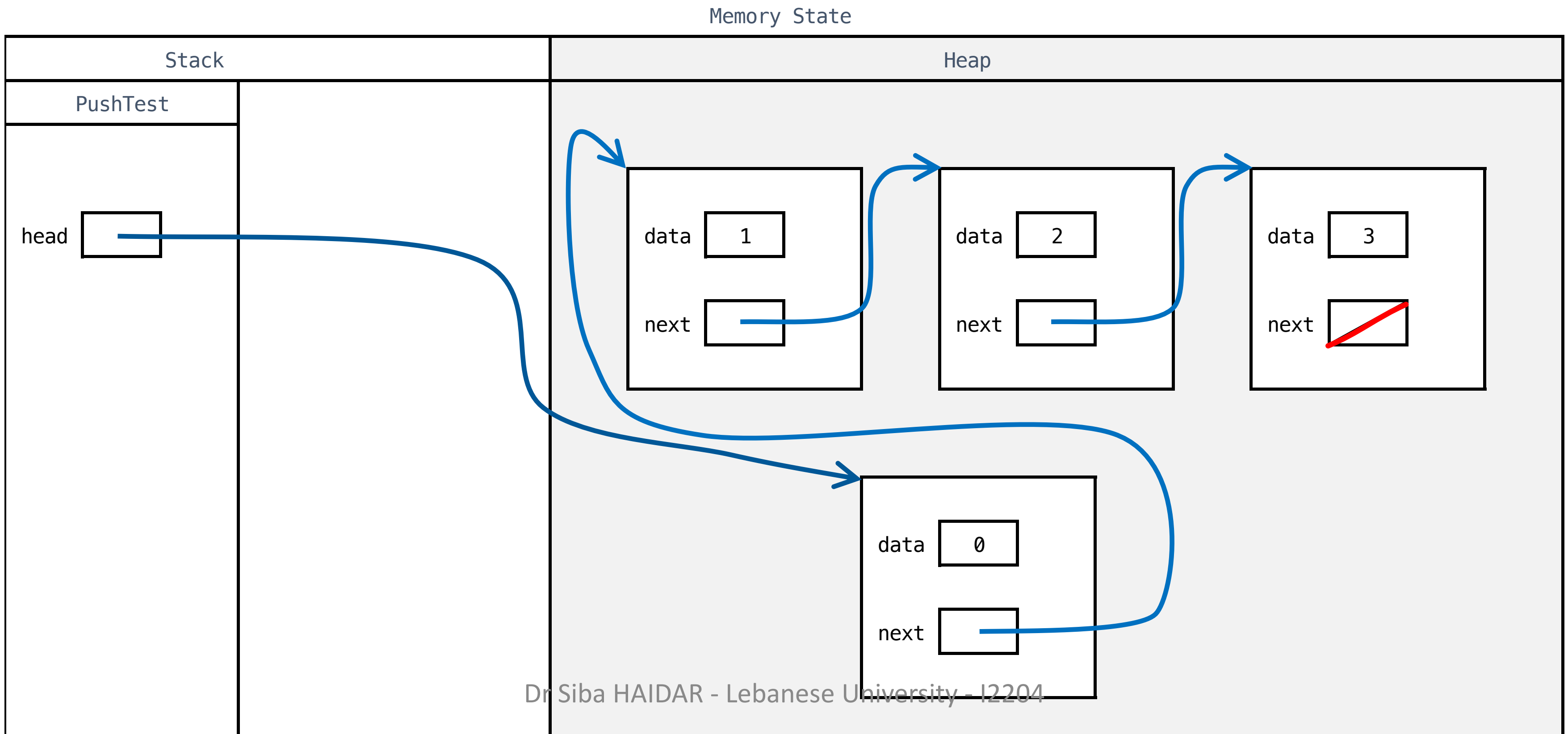
- 3 – link head: `*headRef = newNode;`



memory state

Push Animation (6)

- final state: `{0,1,2,3}`



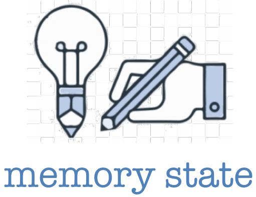
3) Build — At Head With Push()

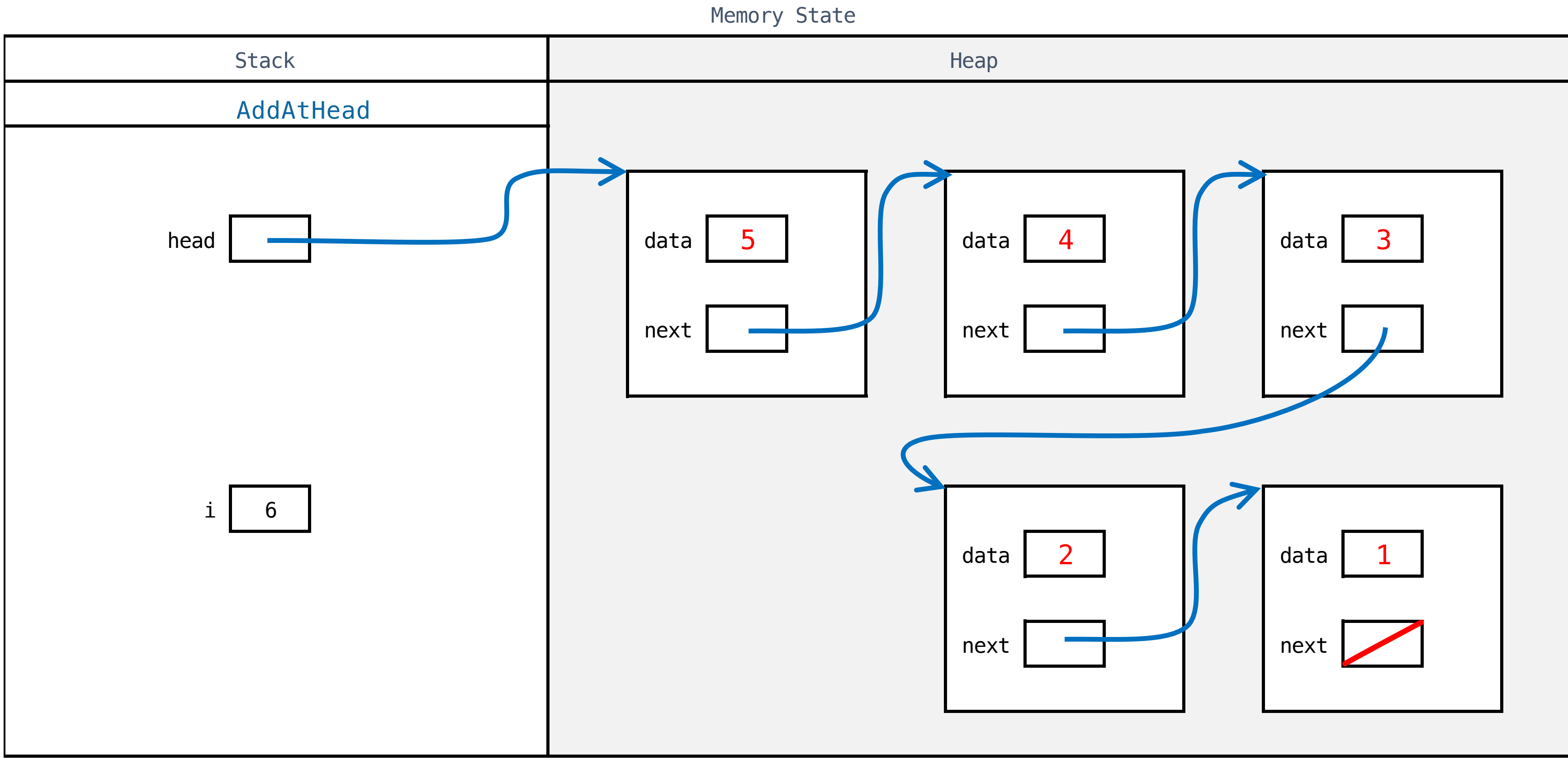
- easiest way to build up a list is by adding nodes at its "head end" with Push()
- code is short and runs fast: lists naturally support operations at head end

disadvantage: elements will appear in the list in reverse order that they are added

```
struct node* AddAtHead() {

  struct node* head = NULL;

  int i;
  for (i=1; i<6; i++) {
    Push(&head, i);
  }

  // {5, 4, 3, 2, 1};
  return head;
}
```

3) Build — At Head With Push()





4) Build — With Tail Pointer

- add nodes at "tail end" of list: locate last node in list, and change its .next field from NULL to point to new node
- one exception is if node is first in list: in that case head pointer itself must be changed

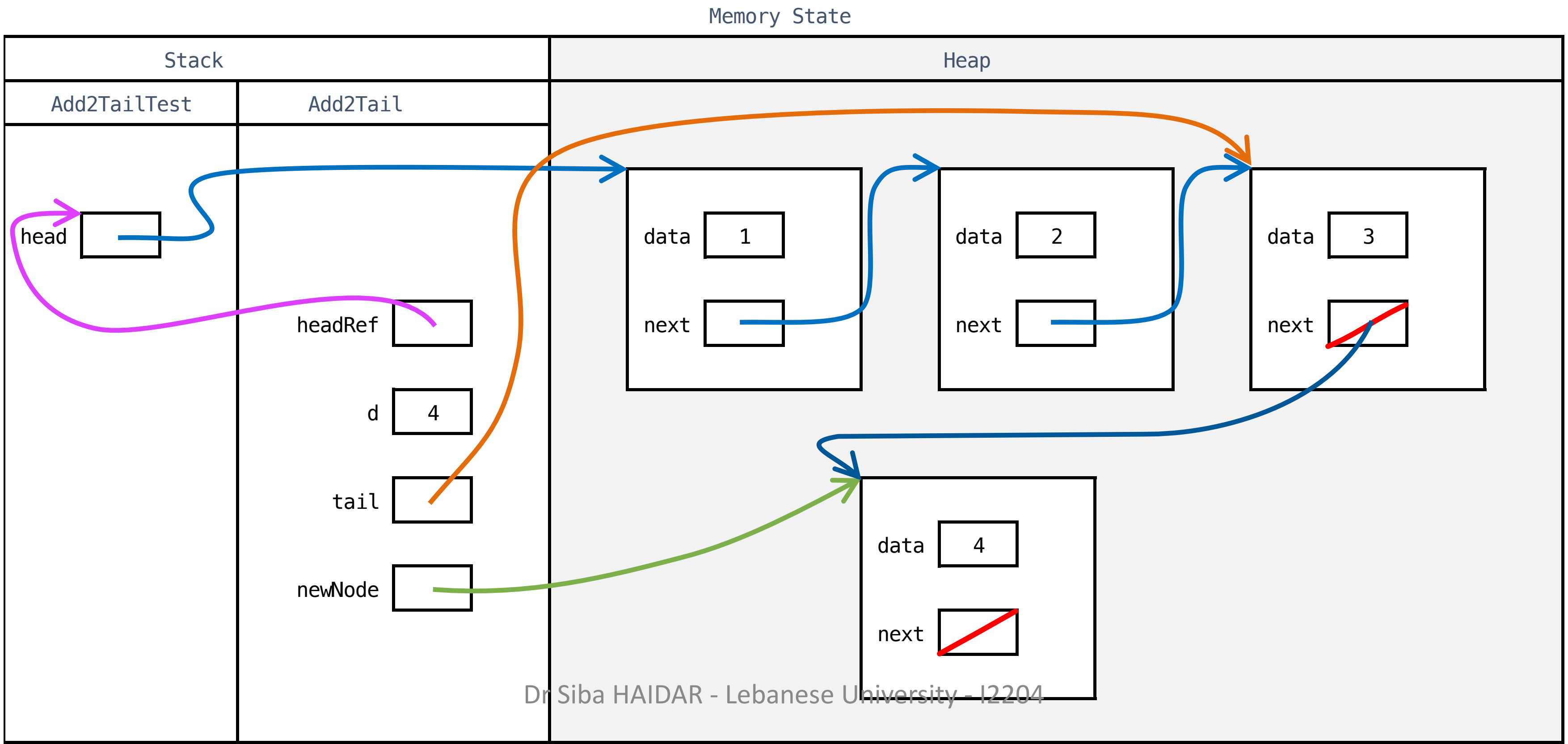
- This is a special case of general rule (insert or delete a node inside a list), for this we need a pointer to node just before that position, then we change its .next field.
- Many list problems include the sub-problem of advancing a pointer to node before point of insertion or deletion.

```
for (current = head;
     current && current->next;
     current = current->next)
  //...
```

4) Build — With Tail Pointer

memory state

example: add 4 to end of {1,2,3}



5) Build — Special Case + Tail Pointer

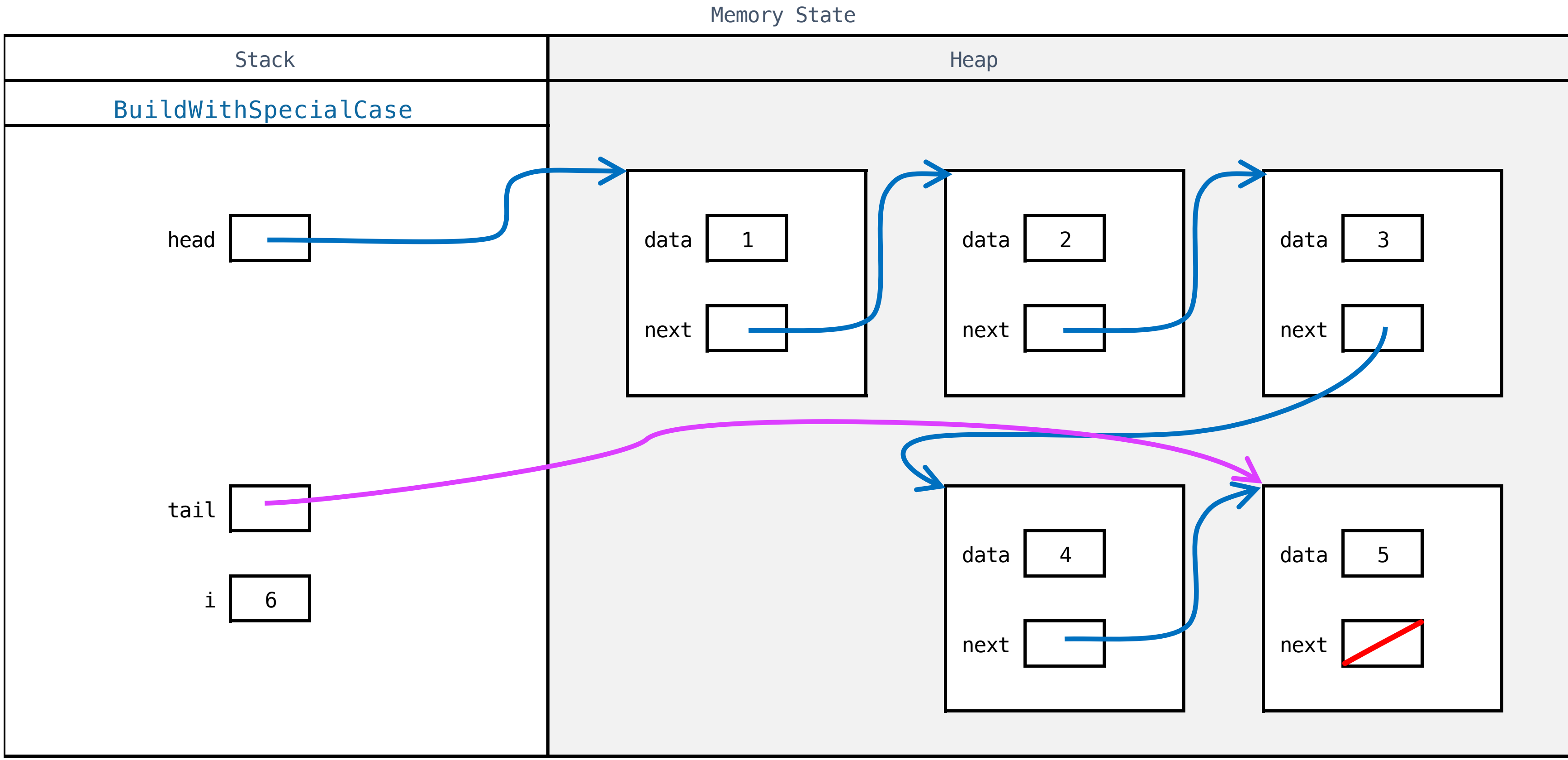
- build up list {1, 2, 3, 4, 5} by appending nodes to tail end
- technique:
  - every first node must be added at head pointer
  - all other nodes inserted after last node using tail pointer

problem: writing separate special case code for first node is unsatisfying

```c
struct node* BuildWithSpecialCase(){
  struct node* head = NULL;
  struct node* tail;
  int i;
  Push(&head, 1);
  tail = head;
  for (i=2; i<6; i++) {
    Push(&(tail->next), i);
    tail = tail->next;
  }
  // {1, 2, 3, 4, 5};
  return head;
}
```

memory state

5) Build — Special Case + Tail Pointer



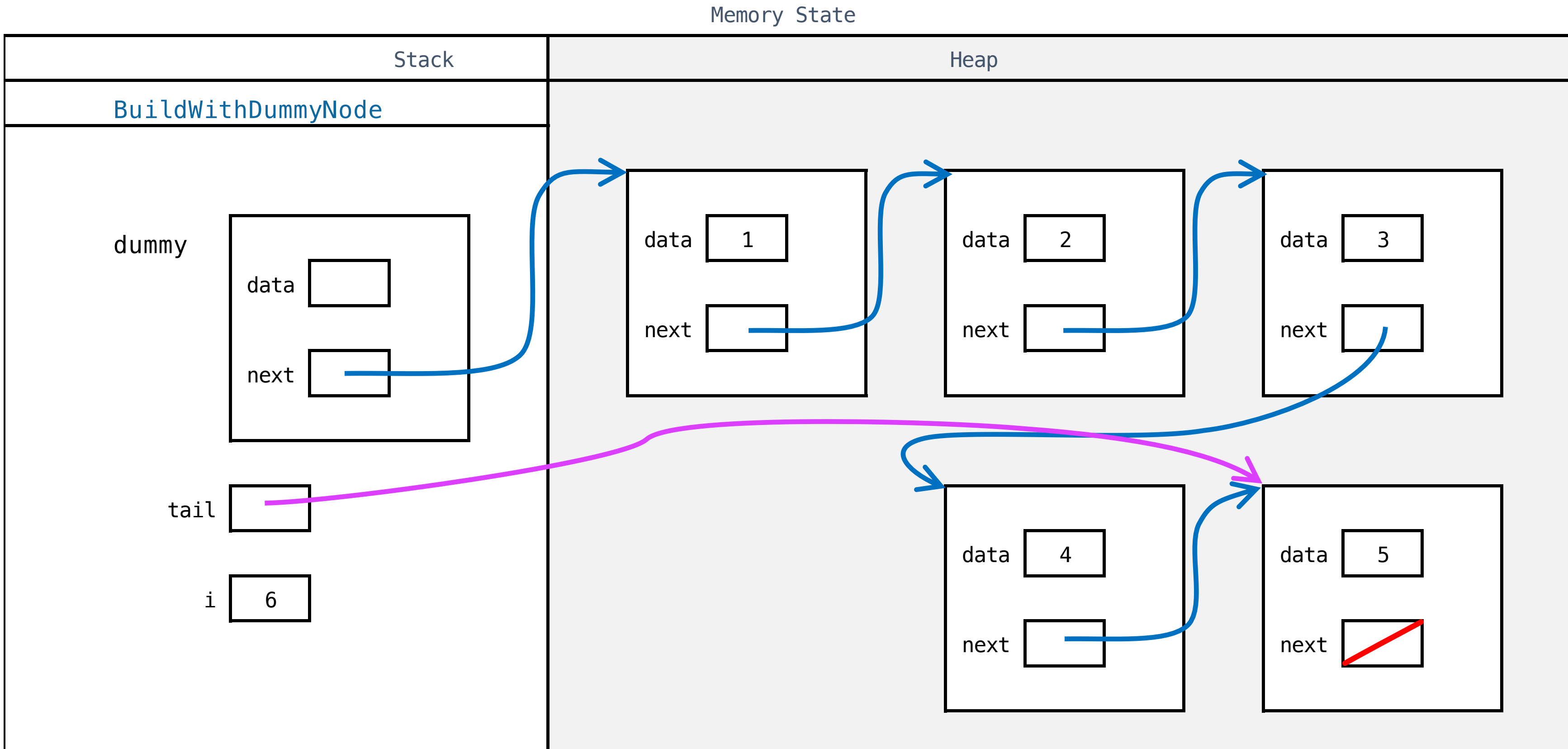
6) Build — Dummy Node

- use temporary dummy node at head of list during computation
- trick with dummy: every node appear to be added after .next field of a node → code for first node is same as for other nodes
- tail pointer plays same role as in previous example, so it also handles first node

```
struct node* BuildWithDummyNode() {
  struct node dummy;
  dummy.next = NULL;
  // Dummy node is temp. first node
  struct node* tail = &dummy;
  int i;
  for (i=1; i<6; i++) {
    Push(&(tail->next), i);
    tail = tail->next;
  }
  return dummy.next;
}
```

memory state

6) Build — Dummy Node



6) Build — Dummy Node

Remarks: can keep dummy node permanent part of list

- empty list is not represented by a NULL pointer
- every list has dummy node at its head
- algorithms skip over dummy node for all operations

- heap allocated dummy node is always present to provide above sort of convenience in code
- dummy-in-the stack strategy, like the example in previous slide, is a little unusual, but it avoids making the dummy permanent part of list

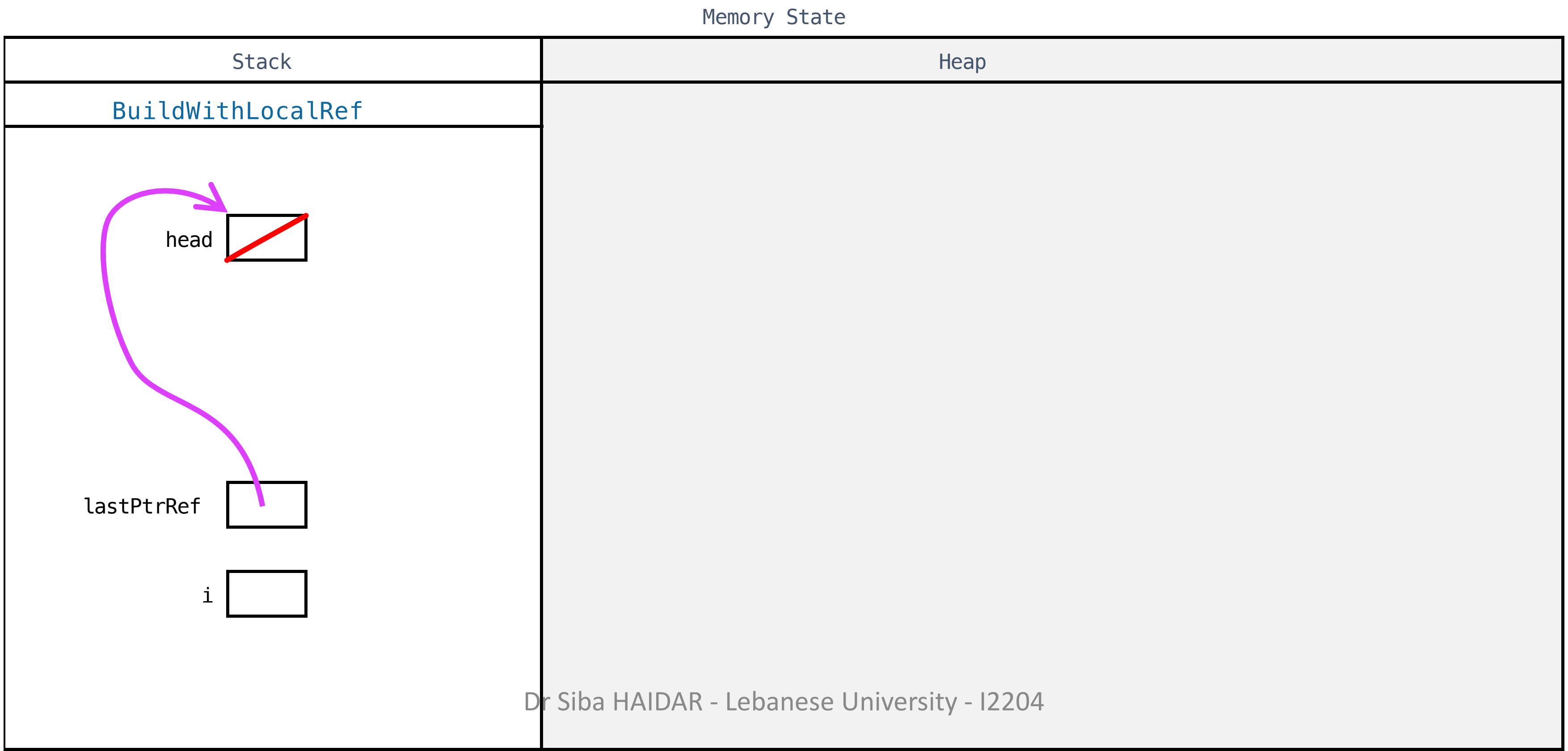
7) Build — Local References

- unify all node cases without using dummy node
- use a local **reference pointer** which always *points to last pointer* in list instead of to last node
- reference pointer starts off pointing to head pointer
- additions to list are made by following reference pointer
- later, it points to .next field inside last node in list

```
struct node* BuildWithLocalRef() {
  struct node* head = NULL;
  struct node** lastPtrRef= &head;

  int i;
  for (i=1; i<6; i++) {
    Push(lastPtrRef, i);

    lastPtrRef= &((*lastPtrRef)->next);

  }
  // head == {1, 2, 3, 4, 5};
  return(head);
}
```

7) Build — Local References

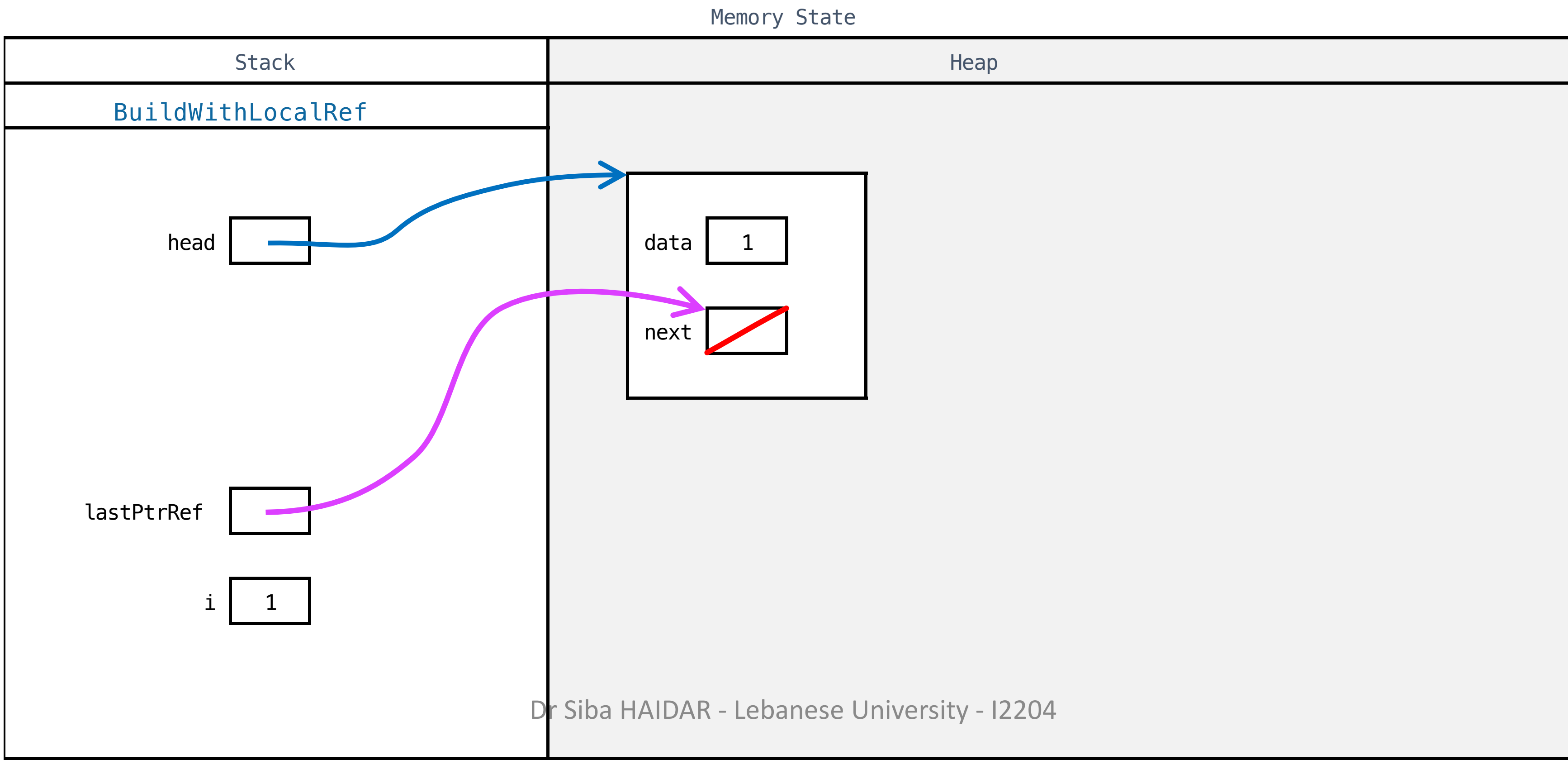
- initial state



memory state

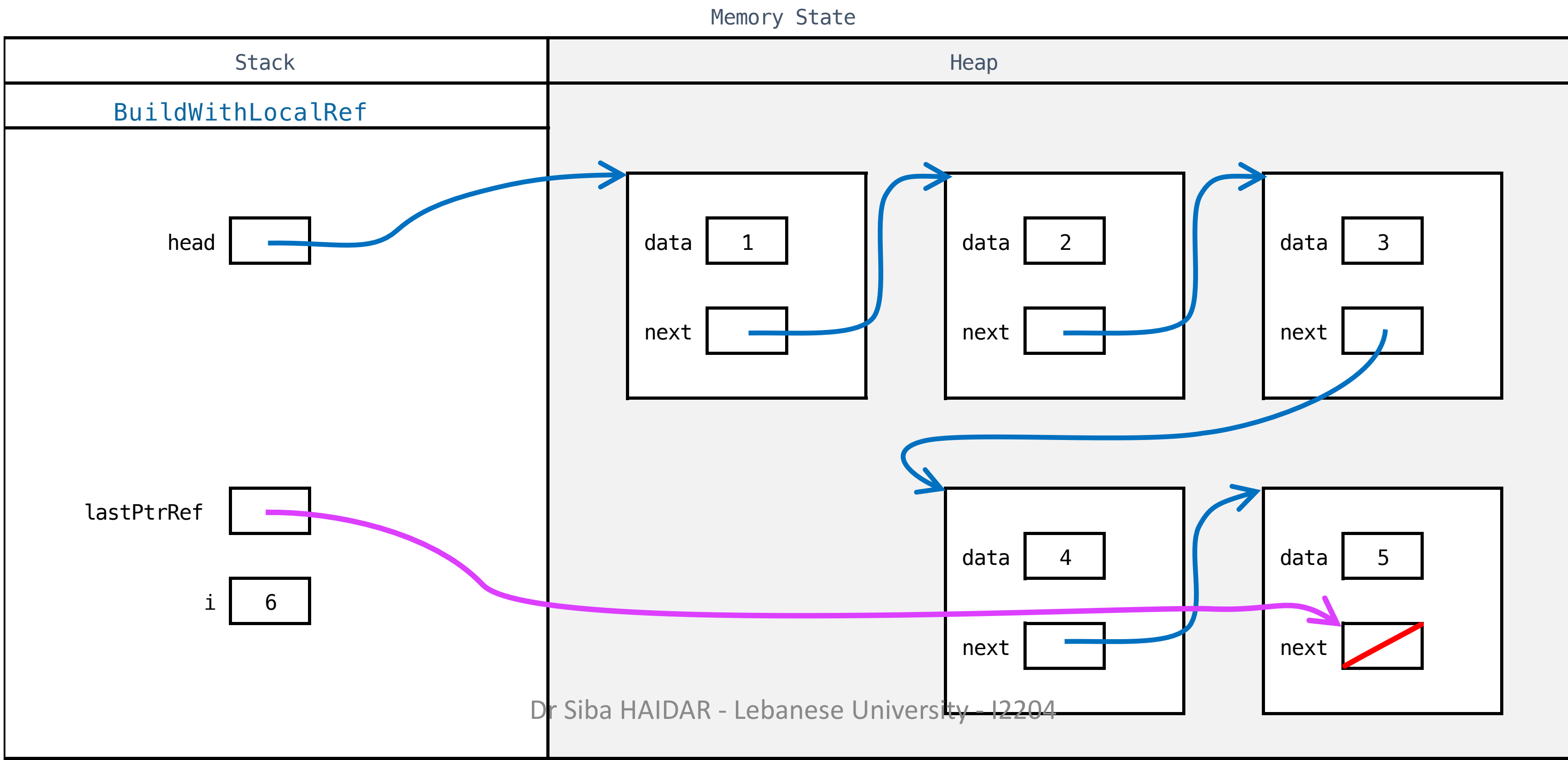
7) Build — Local References

- after first iteration …



7) Build — Local References

- final state



7) Build — Local References

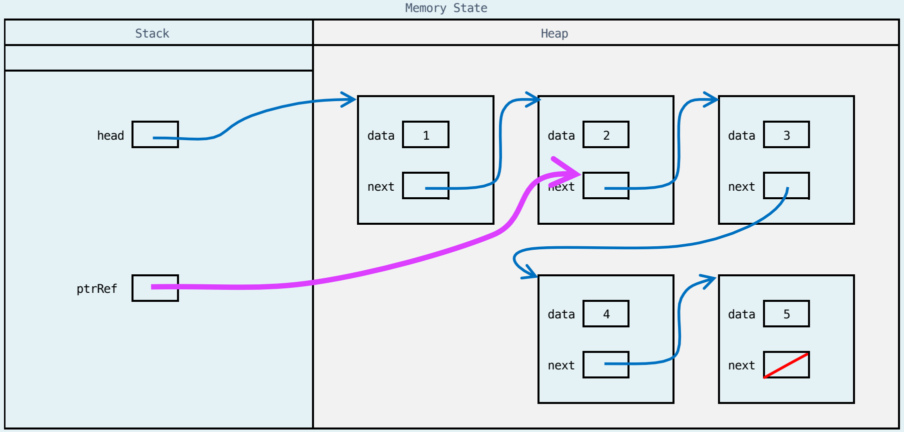

```
struct node* BuildWithLocalRef() {
  struct node* head = NULL;
  struct node** lastPtrRef= &head;

  int i;
  for (i=1; i<6; i++) {
    Push(lastPtrRef, i);

   lastPtrRef= &((*lastPtrRef)->next);

  }
  // head == {1, 2, 3, 4, 5};
  return(head);
}
```

Important Remark about Local References

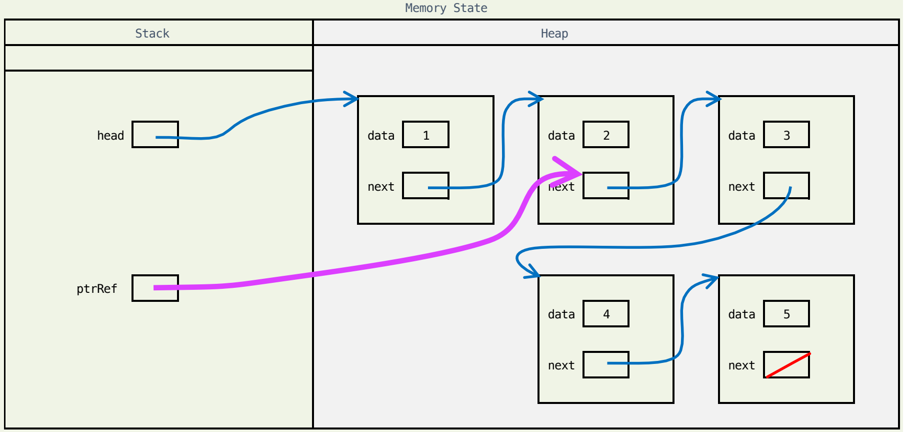
```
node ** ptrRef; // ...
```

Do not confuse 2 different syntaxes → VERY different behavior

```
*ptrRef = (*ptrRef)->next;
```




```
ptrRef = &((*ptrRef)->next);
```



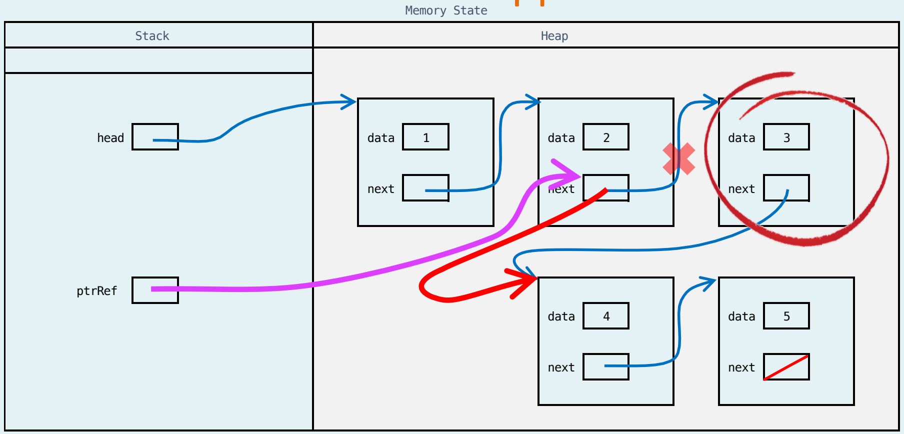
Important Remark about Local References

```
node ** ptrRef; // ...
```

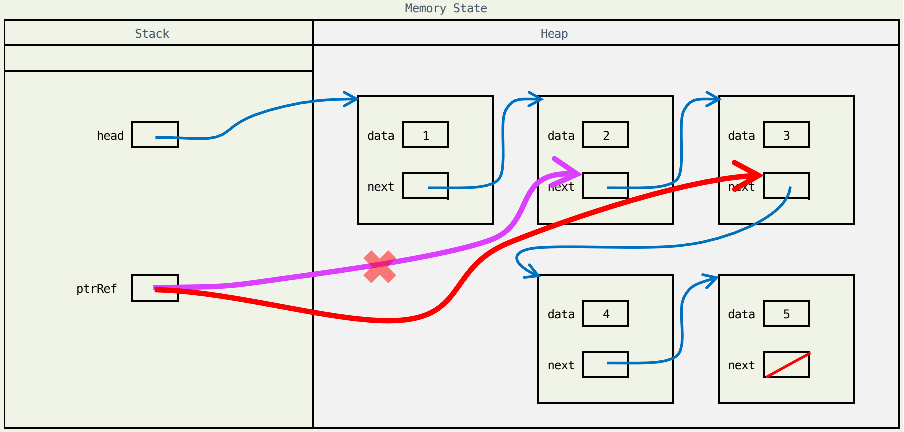
Do not confuse 2 different syntaxes → VERY different behavior

```
node * tmp = *ptrRef;
*ptrRef = (*ptrRef)->next;
free(tmp);
```

→ need to free the skipped node!




```
ptrRef = &((*ptrRef)->next);
```



Linked Lists

1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. **Operations over Linked Lists**
5. Linked Lists Variants

Operations over Linked Lists

1. insert in the middle: insertNth, insertSorted, insertAfter, insertBefore, etc.
2. removeFirst
3. removeLast
4. remove from the middle: …
5. deleteList
6. and many others

- not to forget to free the removed nodes

Study

"Linked List Problems

by Nick Parlante"

Example: InsertNth

- insert a new node at any index within a list
- may specify any index in the range [0..length], and the new node should be inserted so as to be at that index

InsertNthTest()

```
void insertNthTest() {

  // start with the empty list
  struct node* head = NULL;

  insertNth(&head, 0, 13);// {13}

  insertNth(&head, 1, 42);// {13, 42}

  insertNth(&head, 1, 5); // {13, 5, 42}

  deleteList(&head);
  // clean up after ourselves

}
```

InsertNth()

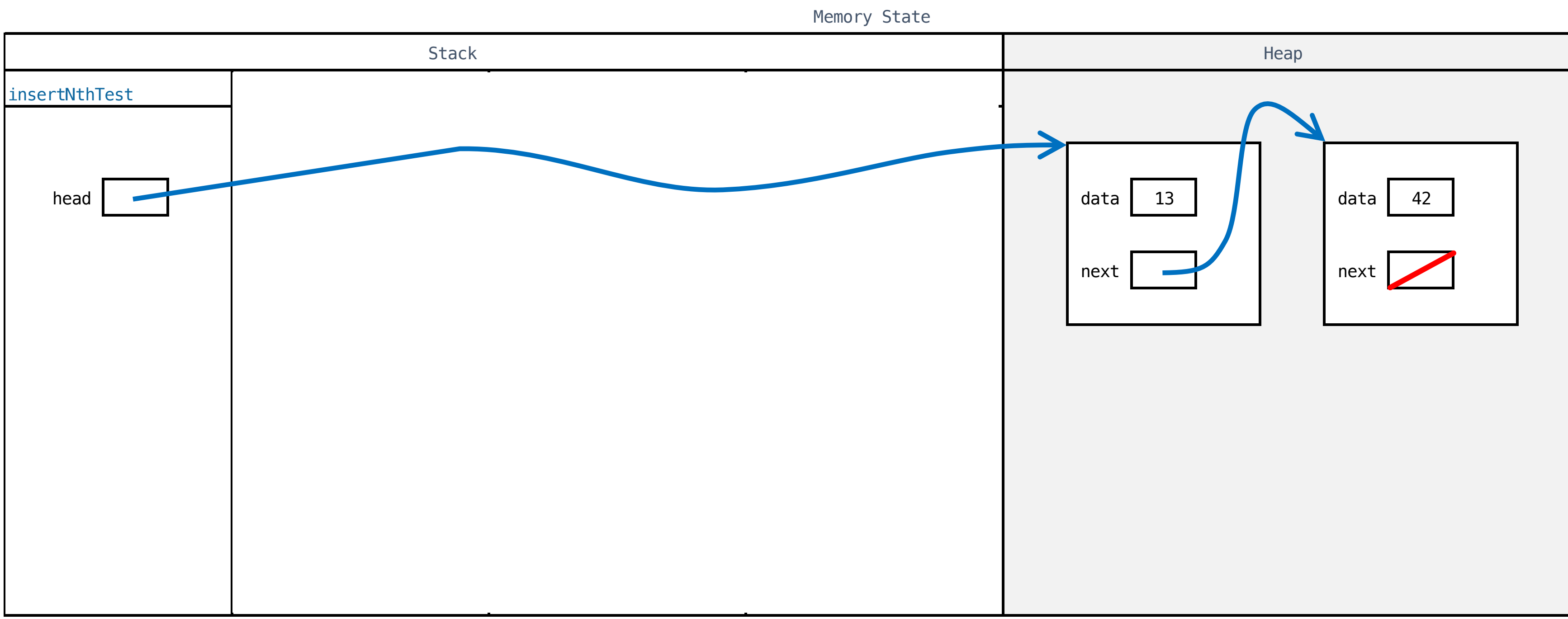
```
void insertNth(struct node** headRef, int index, int data) {

  if( (index < 0) || ( (index > 0) && (*headRef == NULL) ) )
    printf("\nError: insert canceled; index out of range.");

  else if(index == 0)
    Push(headRef, data); //base case is Push

  else
    insertNth( &((*headRef)->next), index-1, data);

}
```

code technique #7

memory state

InsertNth()

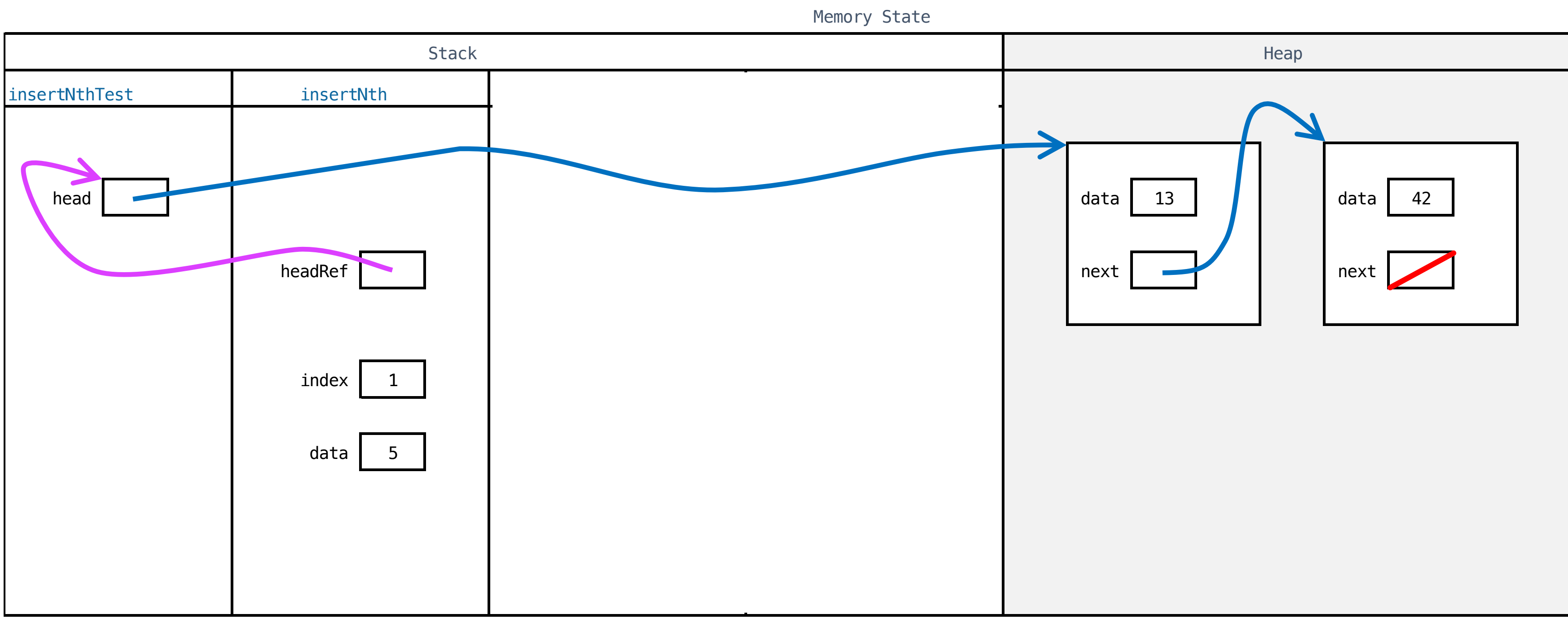
Example: on list `{13,42}`, call `insertNth(&head,1,5)` → `{13,5,42}`



InsertNth()

memory state

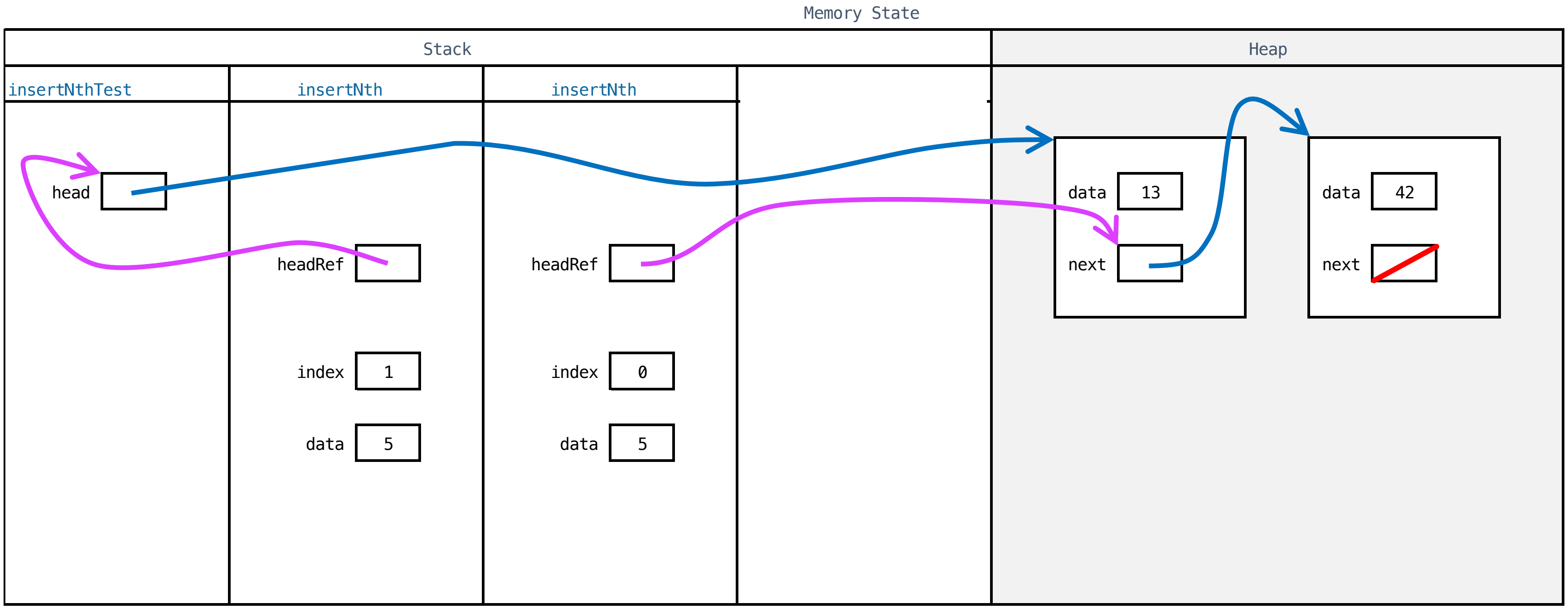
Example: on list {13,42}, call `insertNth(&head,1,5)` → {13,5,42}



memory state

InsertNth()

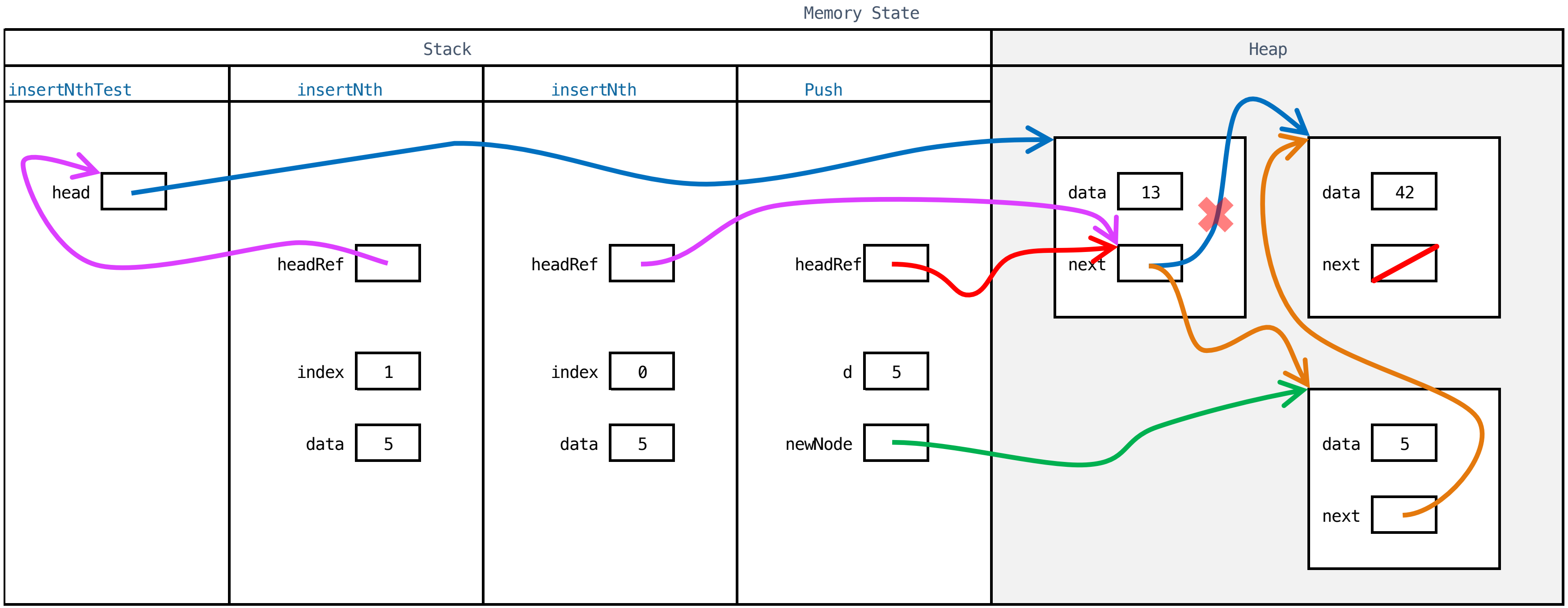
Example: on list `{13,42}`, call `insertNth(&head,1,5)` → `{13,5,42}`



memory state

InsertNth()

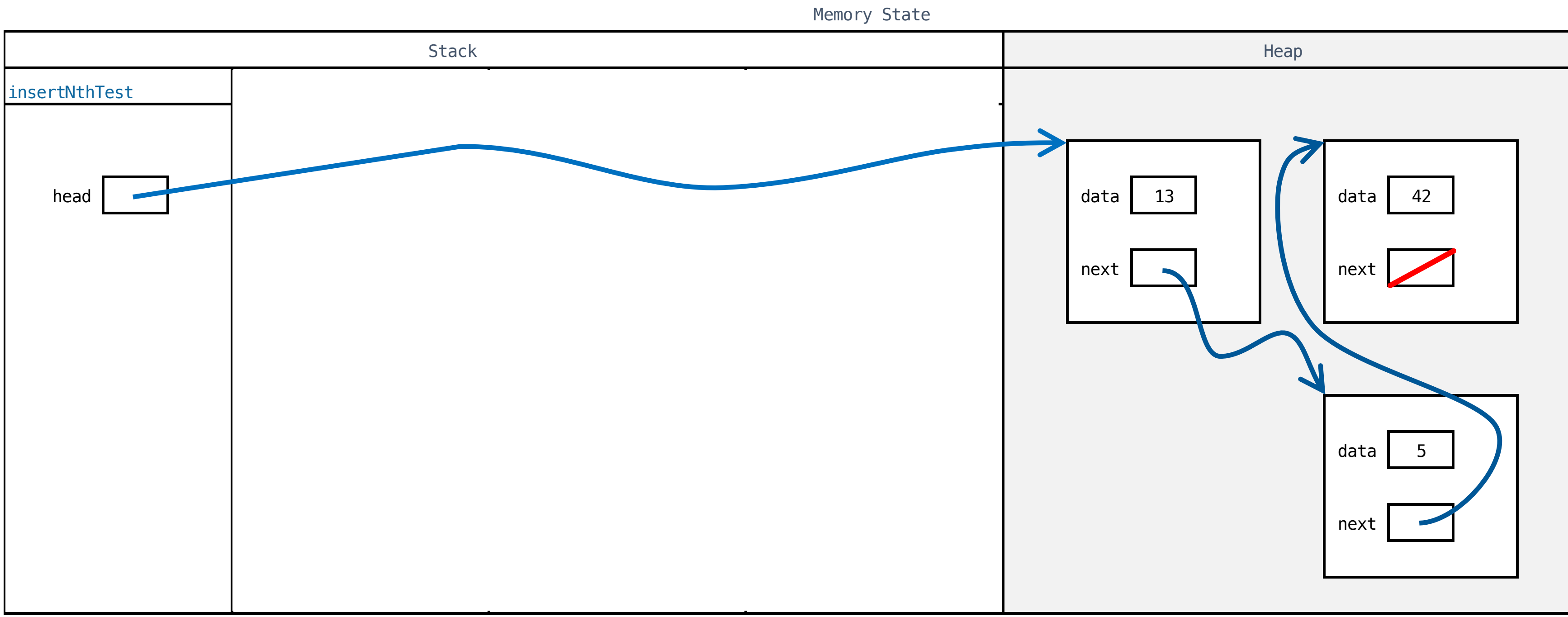
Example: on list {13,42}, call insertNth(&head,1,5) → {13,5,42}



memory state

InsertNth()

Example: on list {13,42}, call `insertNth(&head,1,5)` → {13,5,42}



Linked Lists

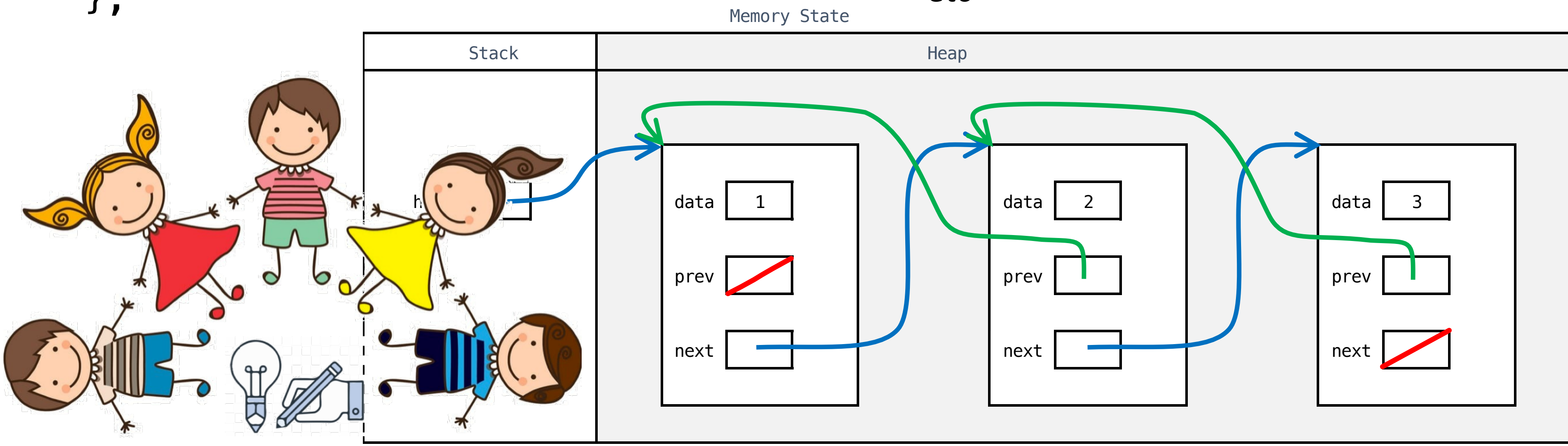
1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. **Linked Lists Variants**

Doubly Linked List (DLL)

- node definition

```c
struct node {
  int data;
  struct node *prev, *next;
};
```

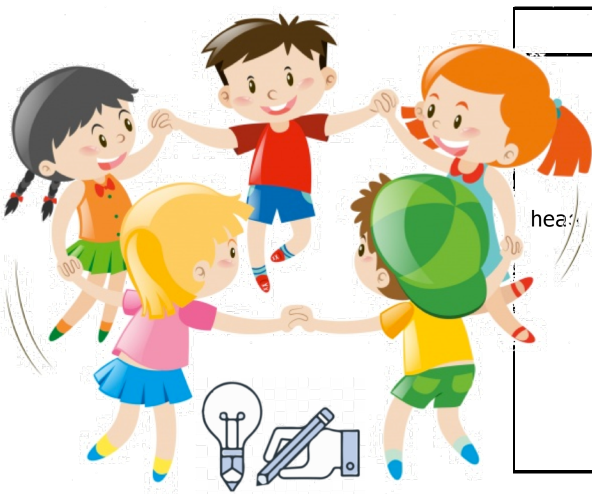
- rethink Push and all the other operations
  - insertNth
  - removeNode,
  - etc



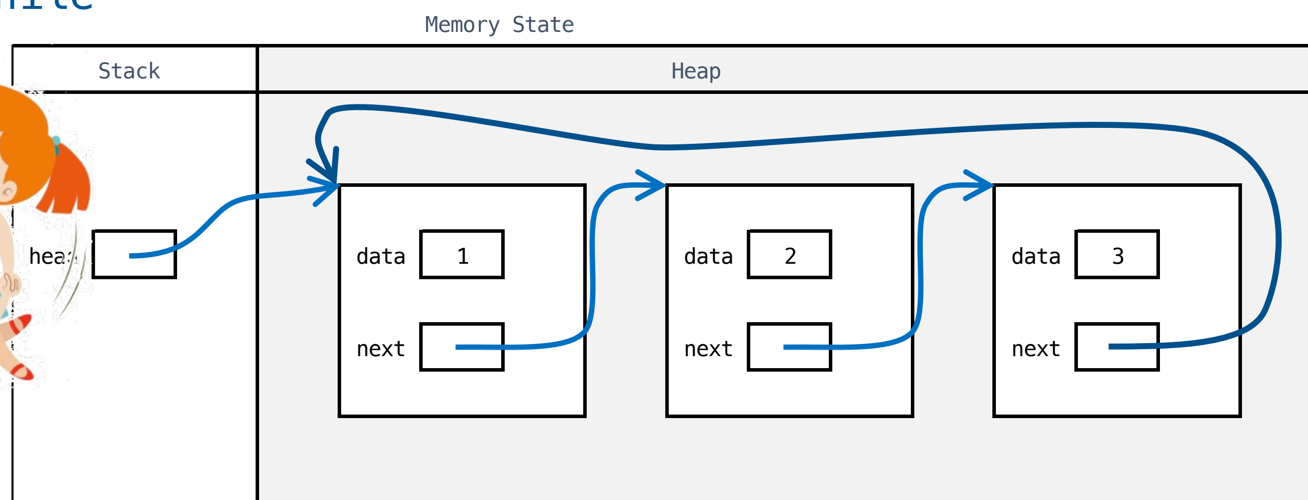
memory state

Circular Linked List (CLL)

- last next field points to the first node
- stopping condition must be changed!
  - use: `do .. while`
- rethink Push and all the other operations
  - insertNth
  - removeNode,
  - etc

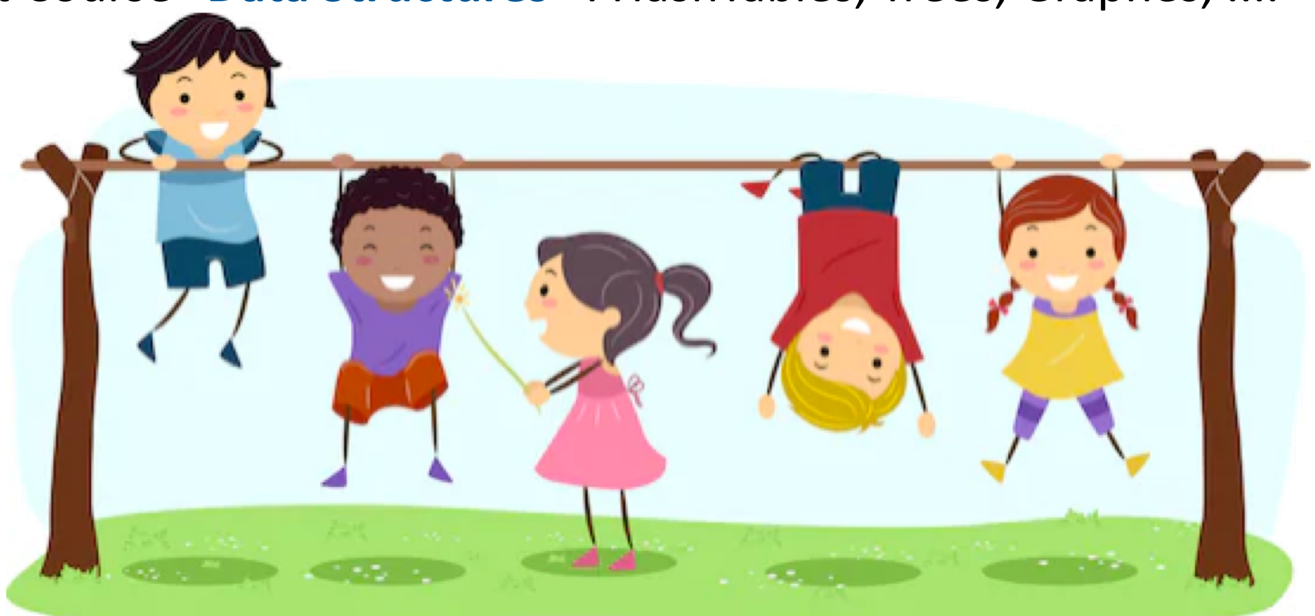


memory state

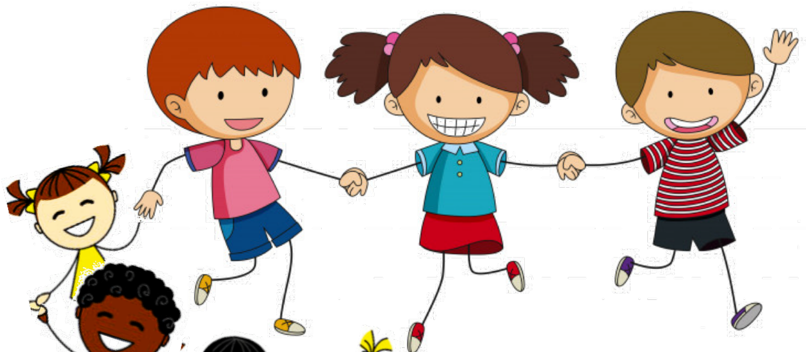


Other Linked Lists

1. Doubly Circular Linked List (DCLL),
2. Linked List with Random Pointer (RLL),
3. Next Course "**Data Structures"** : HashTables, Trees, Graphes, ...



Linked Lists



1. Local vs. Dynamic Memories: Stack & Heap
2. Linked Lists
3. Seven Code Techniques from Nick Parlante
4. Operations over Linked Lists
5. Linked Lists Variants