Inserting Nodes into a Linked List

Kevin Browne

Linked list on the heap

- Data structures tend to be created on the heap, with dynamically allocated memory
- Operations to create, delete, insert and otherwise manipulate the data structure ***at runtime*** require dynamic memory allocation, freeing memory
- These types of operations tend to motivate the usage of data structures altogether

Linked list functions

- Typically data structures are used in conjunction with a set of functions that operate on the data in the data structure, such as:
  - Insert or create elements
  - Delete elements
  - Count elements
  - Search for elements
- In higher-level languages like Java, we're usually given this set of functions via a library
- In C, unless we're using an external library, we make them ourselves!

Linked list functions

- We'll create functions for inserting and deleting nodes from our list
  - With a linked list we could create functions for inserting or deleting nodes **anywhere** in the list
  - We'll create functions for inserting and deleting elements from the head and tail of the list

- Our functions will assume that the list itself could be NULL, i.e. an empty list

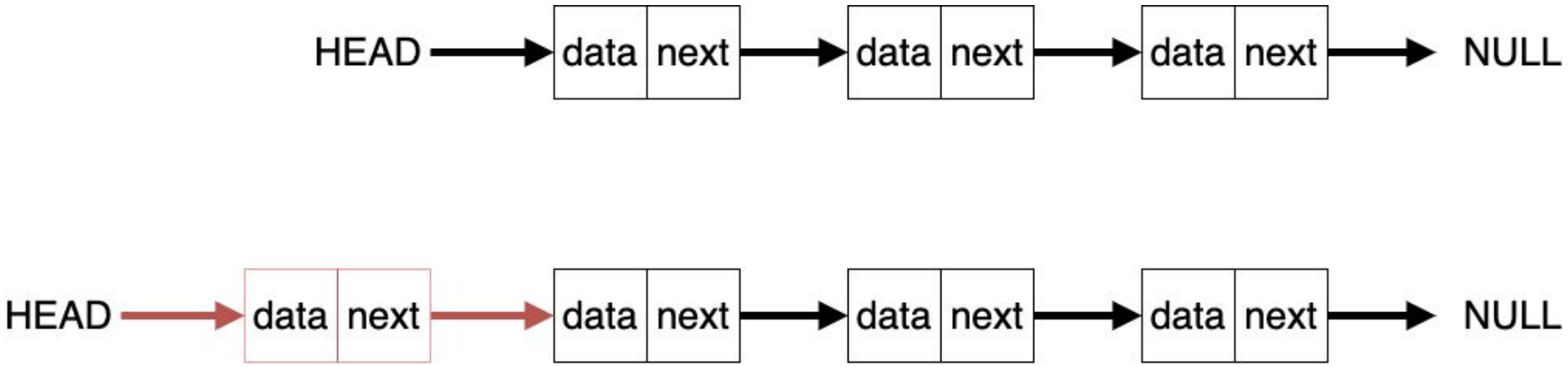
Linked List functions

- We'll create insert functions first!
- It is standard to keep track of a linked list by maintaining a pointer to the head of the list
- So our functions will return a pointer to the head of the new list, given that the operations may change what the head of the list is!

Inserting a new node at the head

- If we're inserting a new node at the head of the list, there are two possibilities:
  - If the list is NULL/blank, the new node we create **is** the entire list, and we would return a pointer to it
  - If the list already has a head, we would create a new node, have it point to the already existing head, and return a pointer to the new node as the new list
- We'll need to dynamically allocate space for the new node, and set its value as well

Before and after insert at head (if the list is not empty/NULL)



Inserting a new node at the head

```
Node* insert_at_head(Node *head, int new_value)
{
  Node *new_node = calloc(1, sizeof(Node));
  new_node->value = new_value;
  if (head == NULL) return new_node;
  else
  {
    new_node->next = head;
    return new_node;
  }
}
```

Now we can insert elements to the head of a list, beginning with an empty/NULL list...

```
Node *list1_head = NULL;

list1_head = insert_at_head(list1_head, 7);
list1_head = insert_at_head(list1_head, 5);
list1_head = insert_at_head(list1_head, 3);

printf("\nPrint out list after inserting at head...\n");
print_list(list1_head);
```

And notice our output, in particular the order of the nodes...

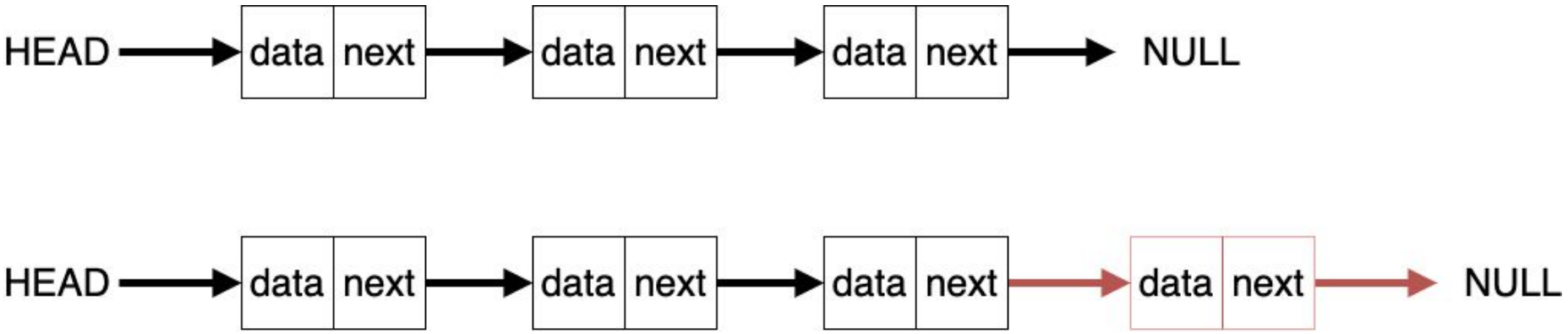
```
Print out list after inserting at head...
Node 0: 3
Node 1: 5
Node 2: 7
```

The last node we inserted is the head!

Inserting new nodes at the tail

- We can also insert new nodes at the tail of the list!
- If the list is NULL/empty, we would just return a pointer to a newly created node
- If the list is not NULL/empty, we would need to traverse the list until we reach the tail node and have the tail node point to our newly created node
  - The tail node is the node that points to NULL

Before and after insert at tail (if the list not empty/NULL)



Inserting a new node at the tail

```
Node* insert_at_tail(Node *head, int new_value)
{
  Node *new_node = calloc(1, sizeof(Node));
  new_node->value = new_value;
  if (head == NULL) return new_node;
  else
  {
    Node *current = head;
    while (current->next != NULL) current = current->next;
    current->next = new_node;
    return head;
  }
}
```

Testing the function...

```
list1_head = insert_at_tail(list1_head, 10);
list1_head = insert_at_tail(list1_head, 12);

printf("\nPrint out list after inserting at tail...\n");
print_list(list1_head);
```

Output:

```
Print out list after inserting at tail...
Node 0: 3
Node 1: 5
Node 2: 7
Node 3: 10
Node 4: 12
```