CMPT 125: Introduction to Computing Science and Programming II

Fall 2023

Week 8: Data structures, revisiting recursion

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Joke of the day

Why do programmers confuse Halloween and Christmas?

Because Oct 31 == Dec 25

3*8 + 1 = 25

2*10 + 5 = 25

Recap from Last Lecture

- Linked Lists
  - A data structure that stores items in a specific way to provide flexible sizes with O(1) time complexity for many operations
  - More efficient implementation of ADTs – using insertFront/insertEnd/removeFront/removeEnd
    - Need to consider the state of the linked lists for different cases (empty/1-item/2+-items)
- Code live demo
  - Using structs and pointers to implement the list
  - Different functions accessing the list via a pointer to it
    - check for null, check for empty/1-item/2+-items
    - use while-loop to traverse

Review from Last Lecture

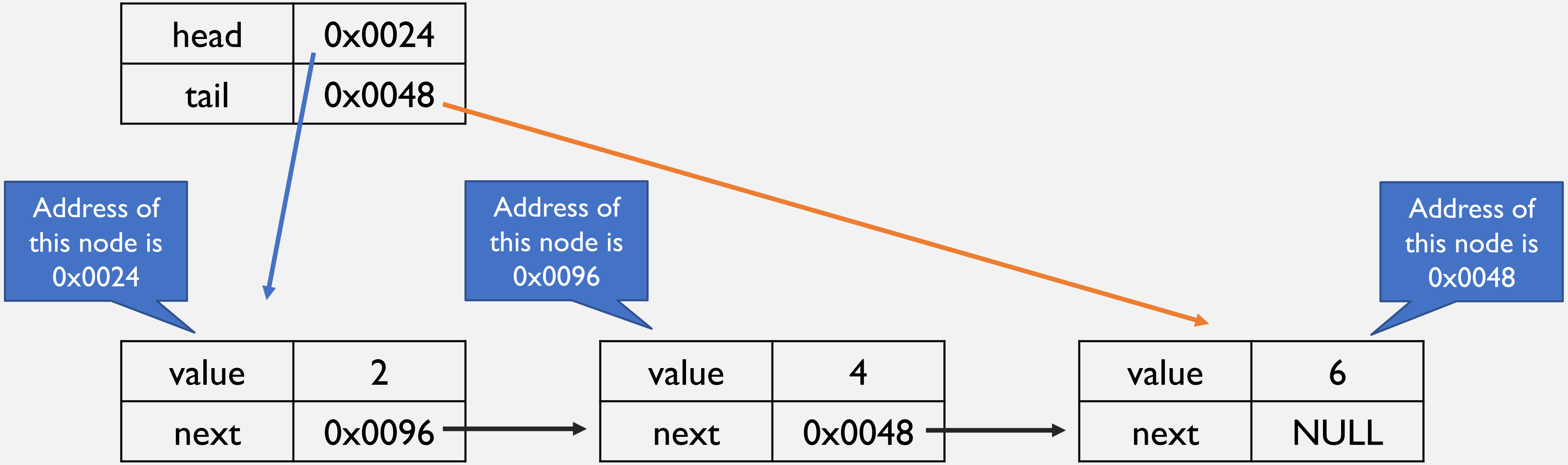
- Implement the rest of the linked list data structure
  - insertEnd, deleteEnd (*you'll need the trick to find the 2nd-last node*)
  - move the code to LinkedList.h and LinkedList.c
    - when using the data structure, include LinkedList.h in the code
- Think about how to insert/delete a node to/from the inside (not head/tail) of a linked list
- Think about how linked list is related to Stacks & Queues

Today

- How to insert/delete a node to/from the inside (not head/tail) of a linked list
  - Cases to consider and be careful with
- Linked list variation
  - Doubly-linked lists
- Revisiting recursion
  - All recursive functions can be implemented without using recursive calls (i.e., non-recursively) using stack

Linked Lists Structure (from Last Lecture)

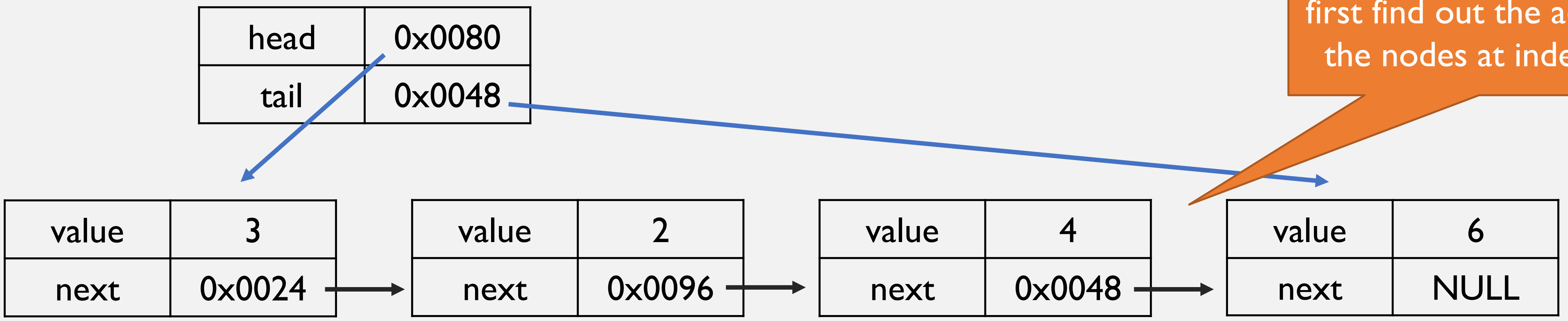
- To store a linked list in C, we add an extra composite data type that works as a header that remembers the first and last node of the list (we call them head & tail)



Inserting An Internal Node (insertAt)

- Typically, a complete implementation of linked list also supports inserting an internal node (neither front nor end)
  - This allows it to support more ADTs
- The steps are a bit more complicated because…
  - we need to change the next pointer of the node after which we are inserting
  - we also need to make the new node point to the node before which we are inserting

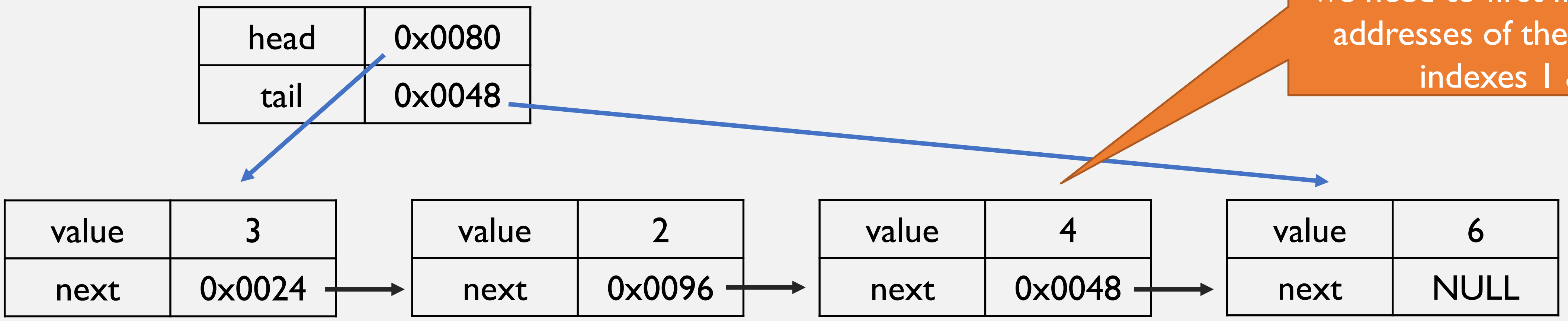
For example, if we want to insert at index 3, we need to first find out the addresses of the nodes at indexes 2 & 3



Removing An Internal Node (removeAt)

- Similarly, a complete implementation of linked list also supports removing an internal node (neither front nor end)
  - This allows it to support more ADTs
- The steps are a bit more complicated because
  - we need to change the next pointer of the node after which we are removing
  - we need to know the address of the node before which we are removing

For example, if we want to remove the node at index 2, we need to first find out the addresses of the nodes at indexes 1 & 3



```c
int LL_length(LList_t* theList) {
    if (theList == NULL) { return -1; }
    else {
        Node_t* current = theList->head;
        int count = 0;
        while (current != NULL) {
            count++;
            current = current->next;
        }
        return count;
    }
}
```

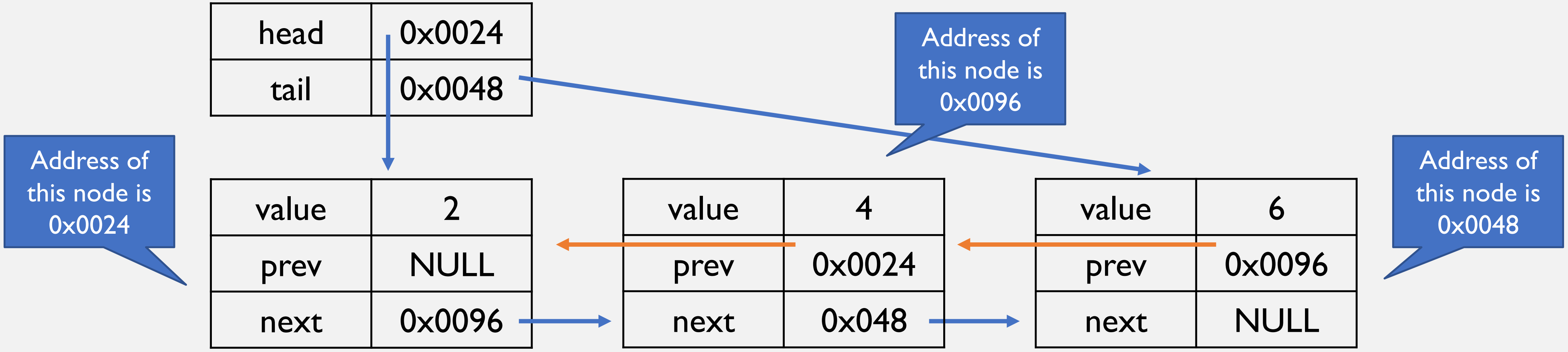
```c
void LL_insertAt(LList_t* theList, int value, int index) {
    if (theList == NULL) { return; }
    else if (index == 0) {//same as LL_insertFront
        LL_insertFront(theList, value);
    } else if (0 < index && index <= LL_length(theList)-1) {//insert inside
        Node_t* newNode = malloc(sizeof(Node_t));
        newNode->value = value;
        newNode->next = NULL;
        Node_t* before = theList->head;
        for (int i=0; i<index-1; i++) { before = before->next; }
        Node_t* after = before->next;
        before->next = newNode; //before now points to the new node
        newNode->next = after; //the new node now points to after
    } else if (index == LL_length(theList)) {//same as LL_insertEnd
        LL_insertEnd(theList, value);
    } else {//beyond current length, treat as invalid index
        return;
    }
}
```

since we know the indexes of the nodes which we need the addresses of, we can use a for-loop with the index as the boundary

LL_removeAt… left as an exercise ☺

Variation: Doubly-Linked List

- Also known as double-linked list, where each node has an extra pointer pointing to its previous node
  - This allows backwards traversal and finding the 2nd-last node in O(1) time (tail->prev), at the cost of an extra space required to store a pointer variable in each node



Implementing Stack ADT with Linked Lists

- A **stack** is a data type that stores a bunch of items and provides access to them in a specific way (LIFO):
  - push: Insert to the **front** of the linked list
  - pop: Remove from the **front** of the linked list
  - isEmpty: check if the linked list is empty (internally done by checking if head/tail is NULL, or via a size variable)
- Think:
  - why inserting & removing in/from the front makes it LIFO?
  - can we achieve the same with the end?

Sample Code for Stack Using Linked Lists

- Push → InsertFront, Pop → RemoveFront, isEmpty → isEmpty

```
LLStack_t* create() {
  LLStack_t* newStack = malloc(sizeof(LLStack_t));
  newStack->data = LL_create();
  return newStack;
}
```

```
typedef struct {
  LList_t* data;
} LLStack_t;
```

```
void push(LLStack_t* theStack, int value) {
  if (theStack == NULL) { return; }
  LL_insertFront(theStack->data, value);
}
```

```
int pop(LLStack_t* theStack) {
  if (theStack == NULL) { return; }
  return LL_removeFront(theStack->data);
}
```

```
int isEmpty(LLStack_t* theStack) {
  if (theStack == NULL) { return 1; }
  return LL_isEmpty(theStack->data);
}
```

```
void free(LLStack_t* theStack) {
  if (theStack == NULL) { return; }
  LL_free(theStack->data);
  free(theStack);
}
```

Implementing Queue ADT with Linked Lists

- A **queue** is a data type that stores a bunch of items and provides access to them in a specific way (FIFO):
  - enqueue: Insert to the **end** of the linked list
  - dequeue: Remove from the **front** of the linked list
  - isEmpty: check if the linked list is empty (internally done by checking if head/tail is NULL, or via a size variable)
- Think:
  - why inserting to the end and removing from the front makes it LIFO?
  - can we achieve the same by reversing the operations?
  - is this approach better than the circular array technique?

Sample Code for Queue Using Linked Lists

- Enqueue → InsertEnd, Dequeue → RemoveFront, isEmpty → isEmpty

```
LLQueue_t* create() {
  LLQueue_t* newQueue = malloc(sizeof(LLQueue_t));
  newQueue->data = LL_create();
  return newQueue;
}
```

```
typedef struct {
  LList_t* data;
} LLQueue_t;
```

```
void enqueue(LLQueue_t* theQueue, int value) {
  if (theQueue == NULL) { return; }
  LL_insertEnd(theQueue->data, value);
}
```

```
int isEmpty(LLQueue_t* theQueue) {
  if (theQueue == NULL) { return 1; }
  return LL_isEmpty(theQueue->data);
}
```

```
int dequeue(LLQueue_t* theQueue) {
  if (theQueue == NULL) { return; }
  return LL_removeFront(theQueue->data);
}
```

```
void free(LLQueue_t* theQueue) {
  if (theQueue == NULL) { return; }
  LL_free(theQueue->data);
  free(theQueue);
}
```

Implementing Dynamic Array ADT with Linked Lists

- A **dynamic array** is a data type that stores a bunch of items and provides access to them in a specific way:
  - setValue: Insert to a specific position of the linked list as indicated by an index
  - getValue: Remove from a specific position of the linked list as indicated by an index
  - isEmpty: check if the linked list is empty (internally done by checking if head/tail is NULL, or via a size variable)
- Think:
  - what happens when the indexed position is already in use during a setValue operation?
  - what happens when the indexed position has no data stored during a getValue operation?

Sample Code for Dynamic Array Using Linked Lists

- SetValue → InsertAt, GetValue → RemoveAt, isEmpty → isEmpty

```
LLDArray_t* create() {
  LLDArray* newArray = malloc(sizeof(LLDArray_t));
  newArray->data = LL_create();
  return newArray;
}
```

```
typedef struct {
  LList_t* data;
} LLDArray_t;
```

```
void setValue(LLDArray_t* theArray, int value, int index) {
  if (theArray == NULL) { return; }
  LL_insertAt(theArray->data, value, index);
}
```

```
int getValue(LLDArray_t* theArray, int index) {
  if (theArray == NULL) { return; }
  return LL_removeAt(theArray->data, index);
}
```

```
int isEmpty(LLDArray_t* theArray) {
  if (theArray == NULL) { return 1; }
  return LL_isEmpty(theArray->data);
}
```

```
void free(LLDArray_t* theArray) {
  if (theArray == NULL) { return; }
  LL_free(theArray->data);
  free(theArray);
}
```

Layered Approach

- Using Linked Lists to build more advanced data structures illustrates a very common approach in CS
  - Allows us to start from simple/basic units and build more complex stuff
  - Allows us to improve our code without affecting the other layers
    - For example, if one day we find a better way to implement Linked List (or even not Linked List), we don't have to re-write Stack/Queue/Dynamic Array as they are just **expecting** a few operations to be available

Revisiting Recursion

Recursion greatly reduces code complexity, but it is not the only way to solve a problem

What Is Recursion Really Doing?

- Recall that recursion needs to establish
  - base case(s) where the problem cannot be made "smaller", if so solve it directly
    - e.g., empty array, n=0, index out of bounds
  - decomposing steps where the problem is divided into smaller problems that can be solved in the same manner
    - e.g., divide into 2 halves and solve 1 half at a time, remove the first/last item, n-1
- The recursive calls can be viewed as subtasks (tasks that are the same in nature but with a smaller "set" to handle)
  - with the order in which they are called remembered
  - which subtask should be returned to when the current subtask is complete
  - with the values they have access to limited to the smaller "set" they are passed with
- It might be tempting to use static/global variables to pass along the smaller "set"
  - a bad idea because they are hard to follow and can break if the same function is invoked several times

Implementing Recursions without Recursive Calls

- Understanding recursive calls as subtasks means – as long as there is a way to remember the order of subtasks, return to the originating subtask when one completes (callbacks), and control access to the data, we don't need to explicitly write a recursive function
- Let's use Quick Sort as an example

```
QuickSort( A[0…N-1] )
  if N == 0
    return
  else
    pivotIndex = partition(A[0…N-1])
    QuickSort(A[0…pivotIndex-1])
    QuickSort(A[pivotIndex-1…N-1])
```

Takes in the input array and does the swaps around a pivot

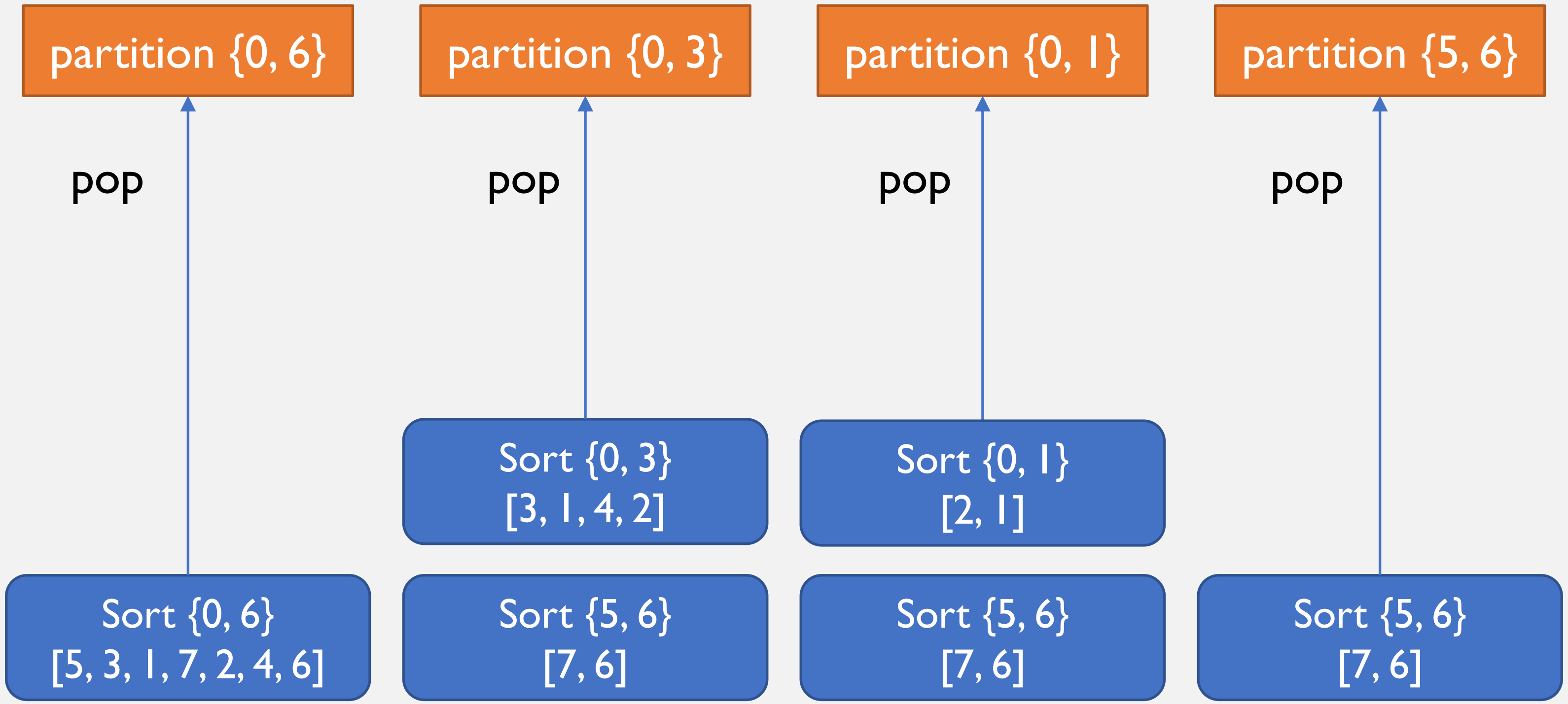
Two recursive calls on smaller "set" of items

Using A Stack to Remember Order & Callbacks

```
QuickSort( A[0…N-1] )
  //each item in the stack S stores the boundaries of the array the subtask can access
  push(S, {0, N-1})
  while (S is not empty)
    {left, right} = pop(S) //current subtask is to sort between indexes “left” to “right”
    pivotIndex = partition(A[0…N-1], left, right) //partition is a non-recursive function that swaps items around pivot
    if pivotIndex+1 < right
      push(S, {pivotIndex+1, right}) //there is at least 2 items to sort on the right half
    if left < pivotIndex-1
      push(S, {left, pivotIndex-1}) //there is at least 2 items to sort on the left half
  end while
```

Quick Sort without Using Recursion

- Sort [5, 3, 1, 7, 2, 4, 6] (N=7)



Exercise: write the content in the array after each partition

Today's Review

- How to insert/delete a node to/from the inside (not head/tail) of a linked list
  - Cases to consider and be careful with
  - Doubly-linked lists
- Revisiting recursion
  - Can greatly reduce complexity of code, but not always the best option (typically has a longer runtime due to creating a sub-routine in the computer memory stack)
    - Can be replaced by using a Stack ADT

Homework!

- Implement the entire linked list data structure
- Implement the Stack, Queue, Dynamic Array ADTs
  - Assume the values are int’s
  - Think: what needs to be changed if we want to store doubles?
- We’ve talked about a variation of linked list being the “doubly-linked list”. Look up other variations
  - We’ll talk about 2 next time