CS 261-020
Data Structures

Lecture 4

Dynamic Array vs. Linked List

Begin Complexity Analysis

1/25/24, Thursday

Oregon State University

Odds and Ends

- Assignment 1 Due Sunday midnight on TEACH
  - No function headers needed if they are already provided
  - Require in-line comments & program header

Recap: C Basics – Function pointers

- When implementing sort() function:
  - This function needs to be able to sort an array of any data type
    - Thus, each element is void*, and we need to use void** to control an array of void*
  - The function needs the size of the array, since it is dynamic
  - The function needs a comparison method to determine which element comes first
    - The method will be provided by the calling function, thus we need to use a function pointer to store the address of that method/function

```
void sort(void** arr, int n, int (*cmp)(void* a, void* b));
```

- Within sort():
  - Whenever we need to compare two values from the array being sorted, we can just call cmp()

```
if (cmp(arr[i], arr[j]) == 0) {
  /* Put arr[i] before arr[j] in the sorted array. */
}
else {
  /* Put arr[i] after arr[j] in the sorted array. */
}
```

Recap: C Basics – Function pointers

- For the calling function (when use sort()):
  - Knows the data type of each element of the array to be sorted
  - Knows the size of the array
  - Knows how to compare the two elements in the array

```
int compare_ints(void* a, void* b) {
  int* ai = a, *bi = b;  /* Cast void* back to int*. */
  if (*ai < *bi)
    return 0;
  else
    return 1;
}
```

  - Function call will be:

```
sort((void**)array_of_ints, number_of_ints, compare_ints);
```

FYI: Using GDB

- Compile with debugging symbols (-g flag), e.g.:

```
gcc -std=c99 filename.c -g -o exe_name
```

- Run it with GDB:

```
gdb ./exe_name
```

FYI: Common GDB Commands

1. `break` – set up break points, e.g.: `b *main` `break 10`
2. `run` – begin execution (until a break point)
3. `print` – see the values of data, e.g. `print i` `print &ptr` `print &main`
4. `next` and `step` – step line by line through the program
5. `continue` – continue until a break point OR the end of the program
6. `backtrace` – prints a backtrace of all stack frame (locate seg fault!!!)
7. x/**100**wx [address or register] – read memory
   - Examine
   - **100** values
   - sized as word (w, 4 bytes)
     - b – byte
     - g – 8 bytes
   - **In hexadecimal (x)**
     - d - decimal

Lecture Topics:

- Dynamic Array (cont.)
- Linked List
- Begin Complexity Analysis

Abstract Data Type (ADT)

- Abstract Data Type (ADT) – a mathematical model for data types
- Specifies:
  - the type of data stored
  - the operations supported on them
  - the types of parameters of the operations.

- Why “abstract”?
  - an implementation-independent view of the data type

Dynamic Arrays

- Elements in an array are stored in a contiguous block of memory
- Allow random access (direct access)
  - i.e., time to access the 1st element = time to access the last element
  - By using array subscript ([]):

```
int* array = malloc(1000 * sizeof(int));
array[0] = 0;
array[999] = 0;
```

Dynamic Arrays (cont.)

- Basic operations:
  - get – Gets the value of the element stored at a given index in the array
  - set – Sets/updates the value of the element stored at a given index in the array
  - insert – Inserts a new value into the array at a given index.
    - Sometimes, dynamic array implementations limit insertion to a specific location in the array, e.g. only at the end.
  - remove – Removes an element at a given index from the array
    - Sometimes, dynamic array implementations avoid moving elements up a spot by only allowing the last element to be removed

Dynamic Arrays (cont.)

- Drawbacks:
  - Fixed size, must be specified when the array is created
    - For static array:

```
int array[50];
```

    - For dynamic array:

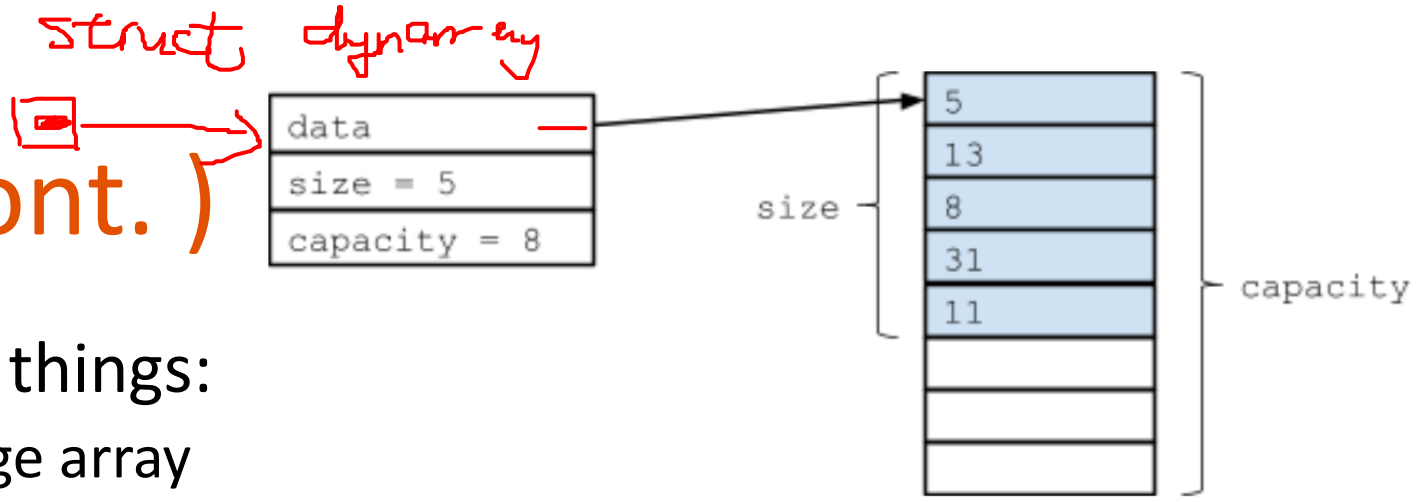
```
int *array = malloc (50 * sizeof(int));
```

→Need to allocate more memory if we need to store more data

  - How?

- Dynamic array DS doesn’t have a fixed capacity
  - Has a variable size and can grow as needed

Dynamic Arrays (cont.)



- Need to keep track of three things:
  - data – underlying data storage array
  - size – number of elements currently stored in the array
  - capacity – number of elements data has space for before it must be resized
- How it works?
  - An array of known capacity is maintained by the dynamic array DS.
  - As elements are inserted, they are simply stored in data
  - If an element is inserted into the dynamic array, and there isn't capacity for it in the underlying data storage array (data), the capacity of the underlying data storage array is doubled. Then the new element is inserted into this larger data storage array.

Dynamic Arrays

s: 0 c: 2

s: 1 c: 2

5	

s: 2 c: 2

5	8

s: 3 c: 4

5	8	1	

s: 4 c: 4

5	8	1	4

s: 5 c: 8

5	8	1	4	9			

s: 6 c: 8

5	8	1	4	9	0		

s: 7 c: 8

5	8	1	4	9	0	6	

s: 8 c: 8

5	8	1	4	9	0	6	7

s: 7 c: 8

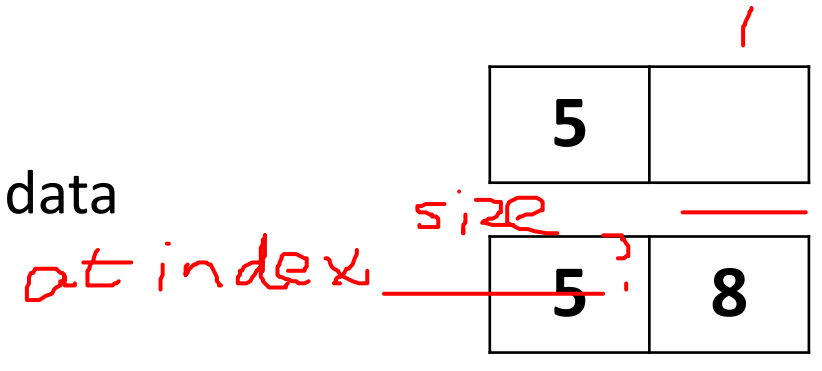
5	8	4	9	0	6	7	

s: 6 c: 8

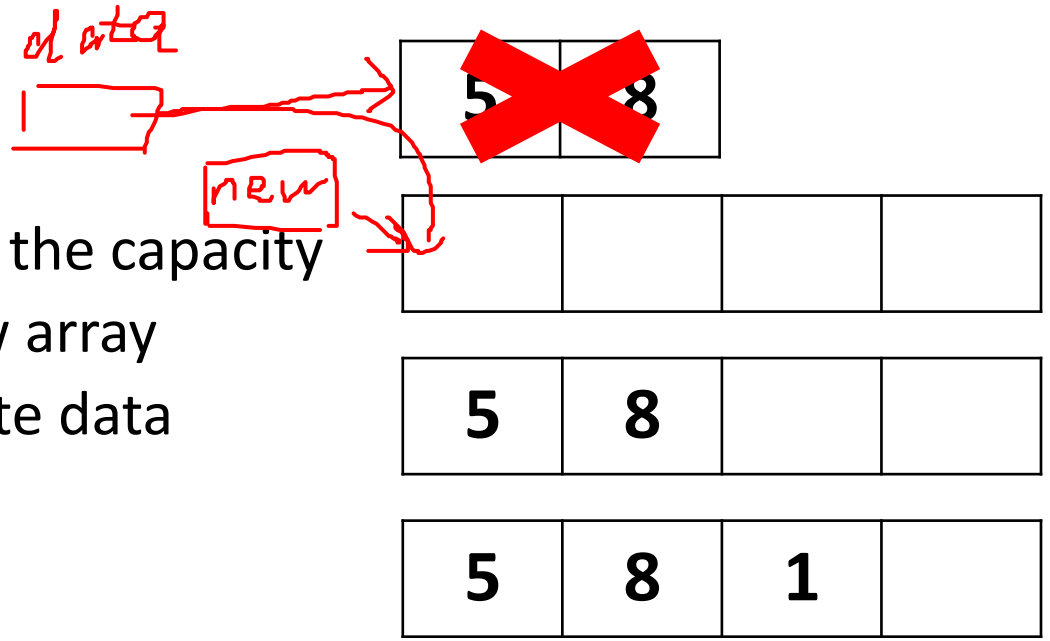
5	8	4	9	6	7		

Inserting an element into dynarray

- Case 1: if size < capacity
  - At least one free spot in data
  - Insert the new element

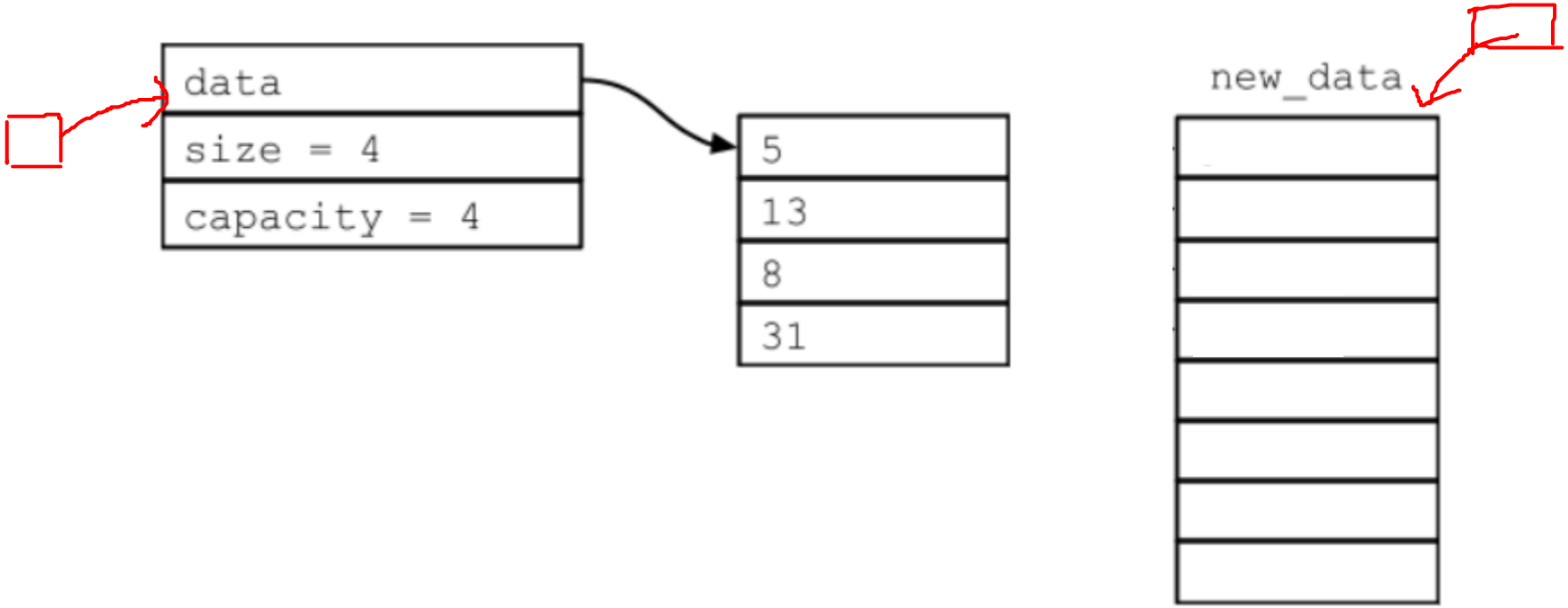


- Case 2: if size == capacity
  - No free spot in data
  - Step 1: allocate a new array that has twice the capacity
  - Step 2: copy all elements from data to new array
  - Step 3: delete the old data array and update data
  - Step 4: Insert the new element



Another Example

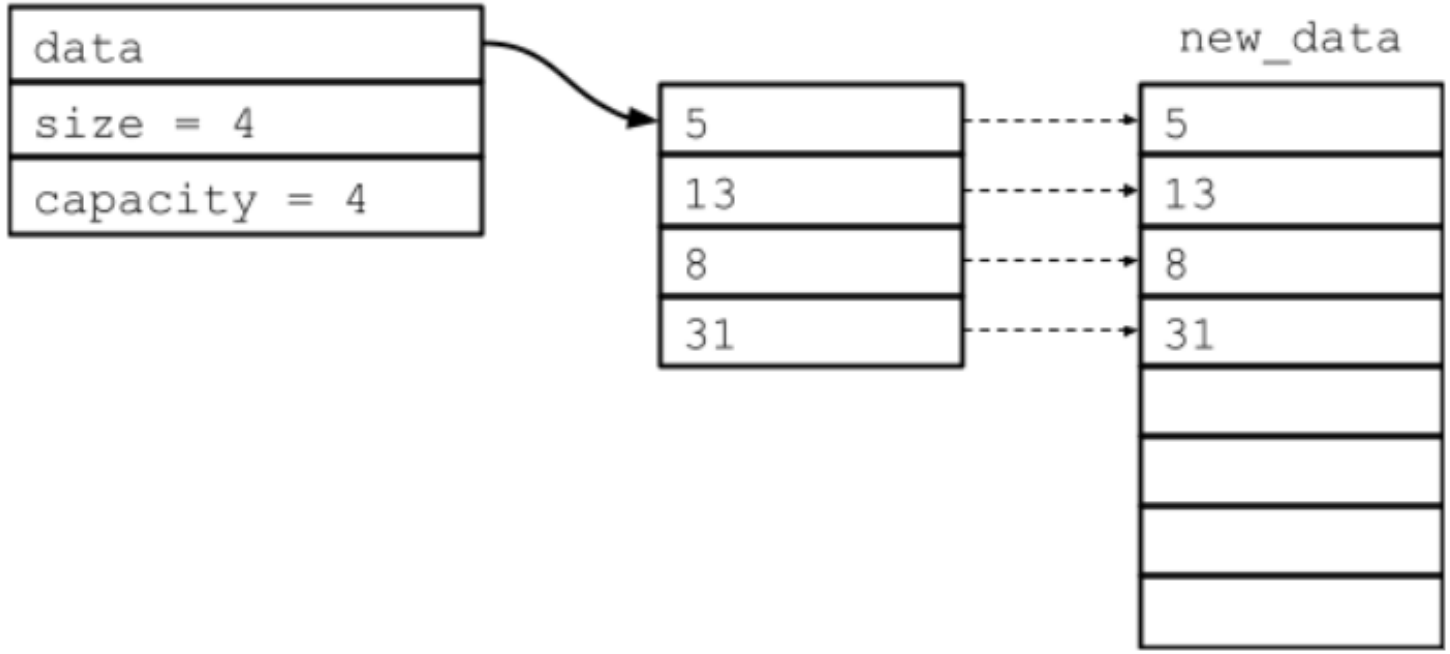
- Insert 16 to the following dynamic array:



- Step 1: allocate a new array that has twice the capacity

Another Example

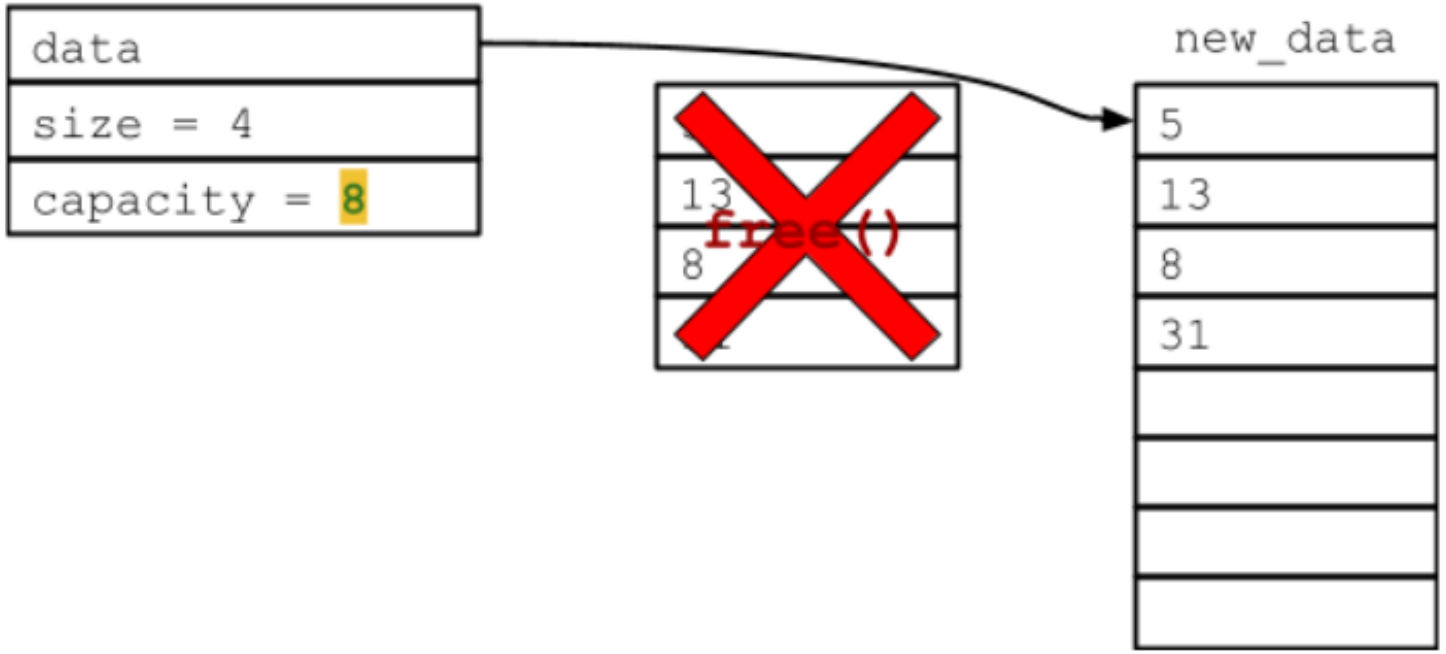
- Insert 16 to the following dynamic array:



- Step 2: copy all elements from data to new array

Another Example

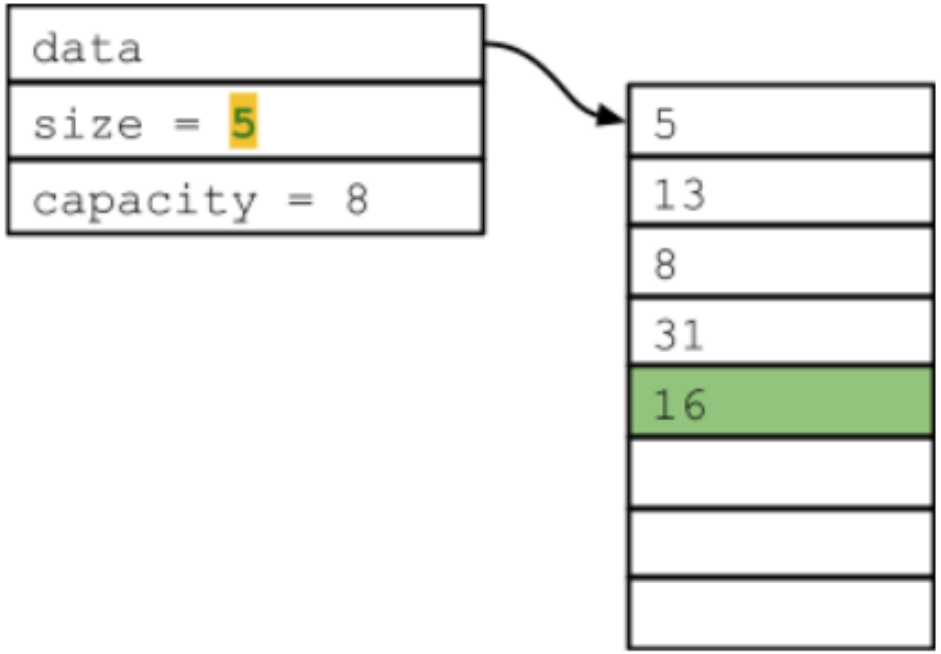
- Insert 16 to the following dynamic array:



- Step 3: delete the old data array and update data

Another Example

- Insert 16 to the following dynamic array:



- Step 4: Insert the new element

Common Mistakes

- 1. dynarray_create():
  - In order to manage a dynamic array, how many `struct dynarray` do you need?

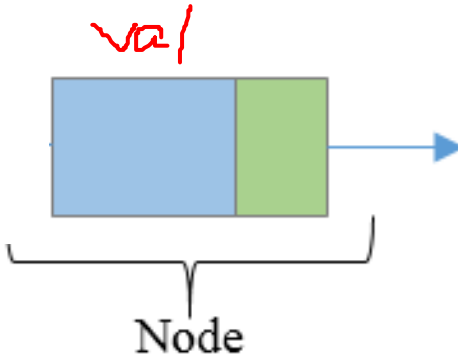
- 2. dynarray_insert():
  - When size == capacity, do you need to free the entire `struct dynarray`, i.e., `free(da),` before resizing?

Lecture Topics:

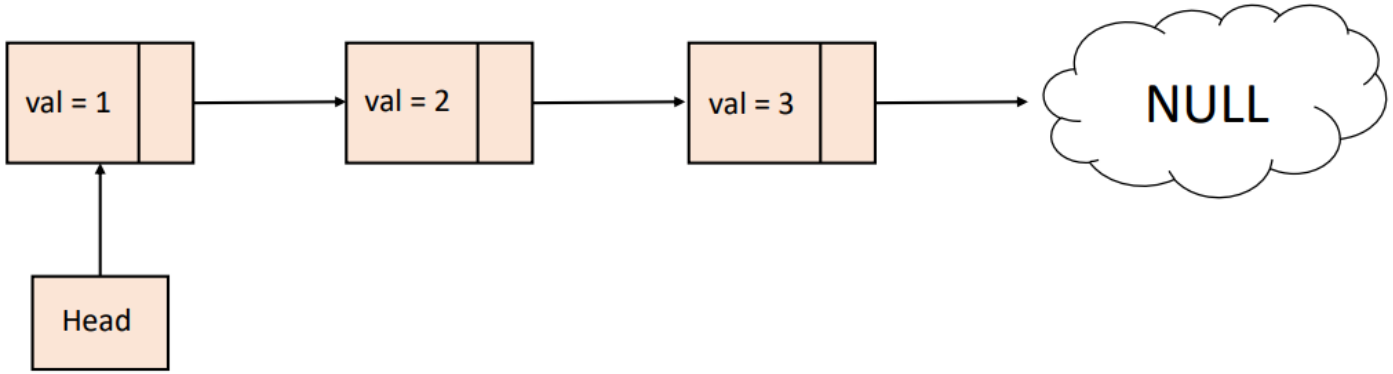
- Dynamic Array (cont.)
- Linked List
- Begin Complexity Analysis

Linked List

```
struct node {
        void* val;
        struct node* next;
};
```



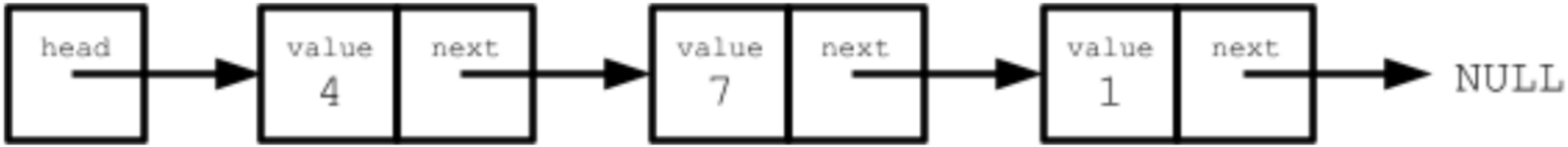
- Linear Data Structure
- Elements in a linked list are stored in nodes and chained together
  - Not in contiguous memory
  - Thus, no random access
- A linked list in which each node points only to the next link in the list is known as a singly-linked list.
  - E.g.:



Linked List

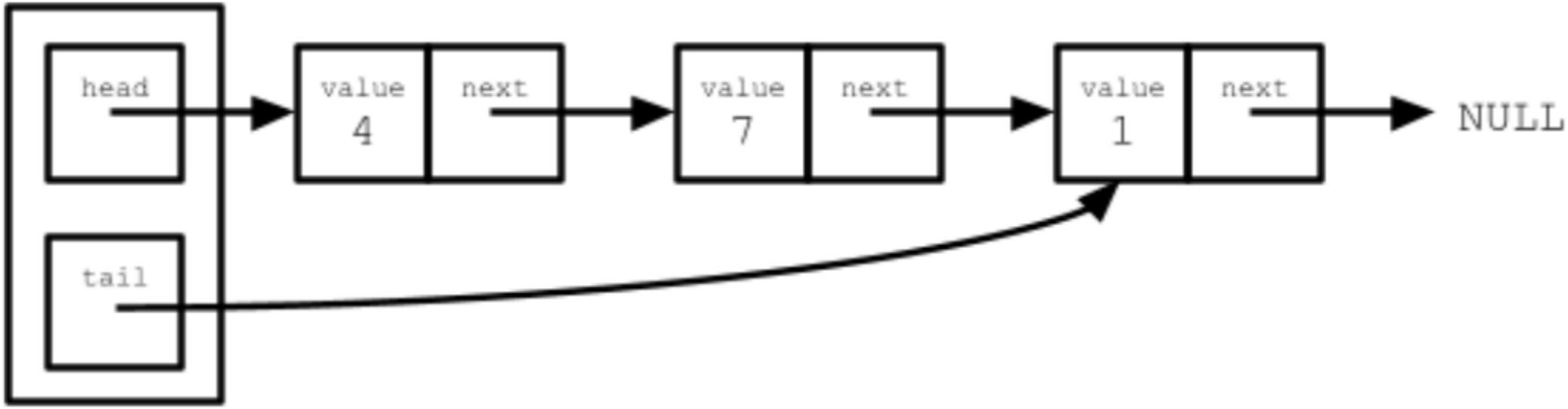
- Always contains as many nodes as it has stored values
  - Add an element → allocate a node, add it to the list
  - Remove an element → free the node from the list

- Many forms of linked list:
  - Keeps track only of the first element in the list, known as head



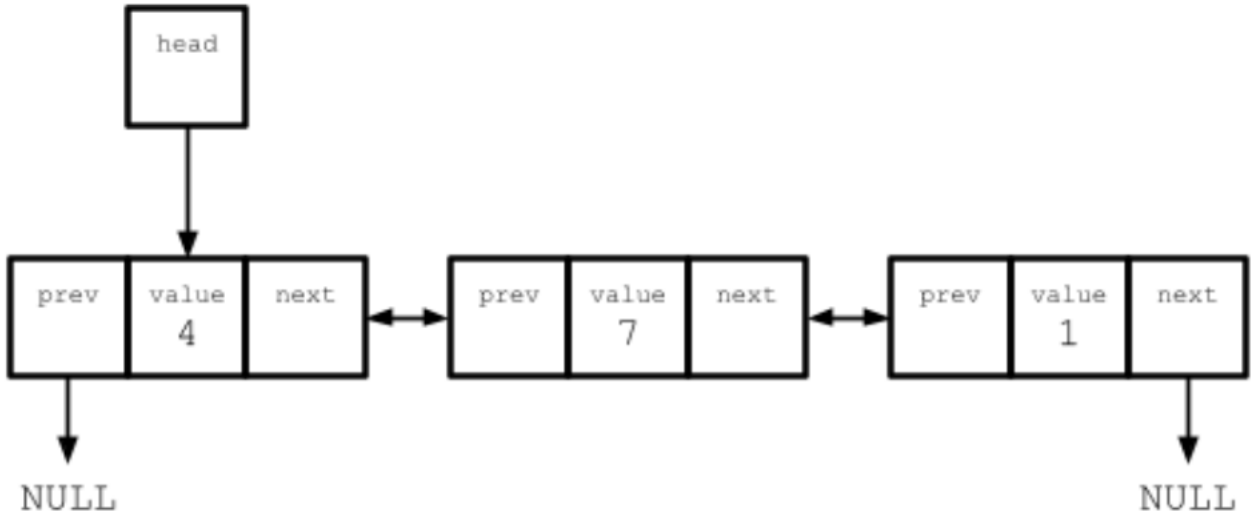
Linked List

- Many forms of linked list:
  - Keeps track only of the first element in the list, known as head
  - Keeps track of both the head of the list and the tail, or last element



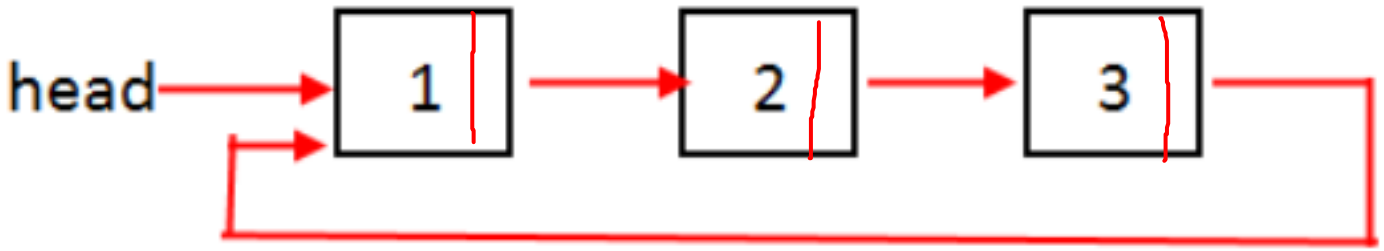
Linked List

- Many forms of linked list:
  - Keeps track only of the first element in the list, known as head
  - Keeps track of both the head of the list and the tail, or last element
  - Each node keeps track of both the *next* link and the *previous* link in the list, known as a doubly-linked list



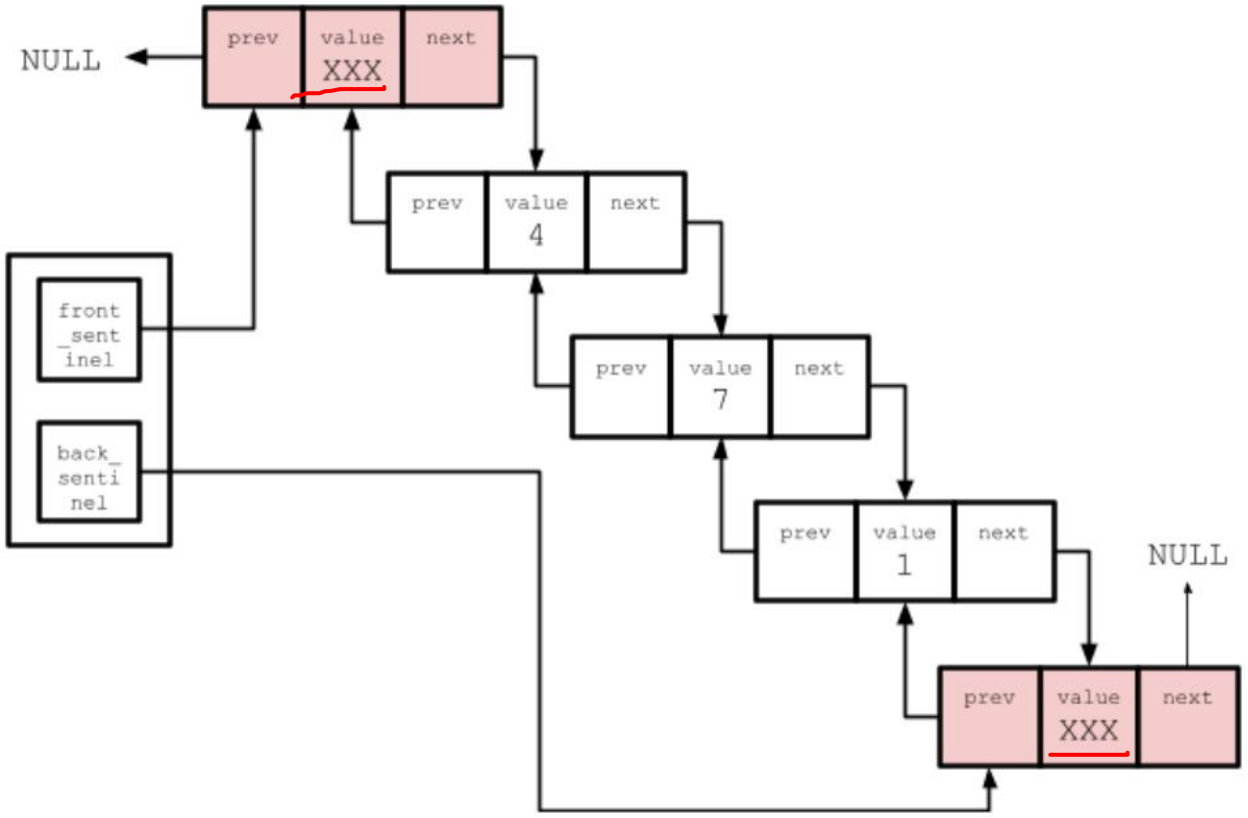
Linked List

- Many forms of linked list:
  - Keeps track only of the first element in the list, known as head
  - Keeps track of both the head of the list and the tail, or last element
  - Each node keeps track of both the *next* link and the *previous* link in the list, known as a doubly-linked list
  - Last node points to the first node, known as circular-linked list



Linked List

- Many forms of linked list:
  - With sentinels, which are special nodes to designate the front/end of the list
    - E.g.: a doubly-linked list using both front and back sentinels



Inserting an element into linked list

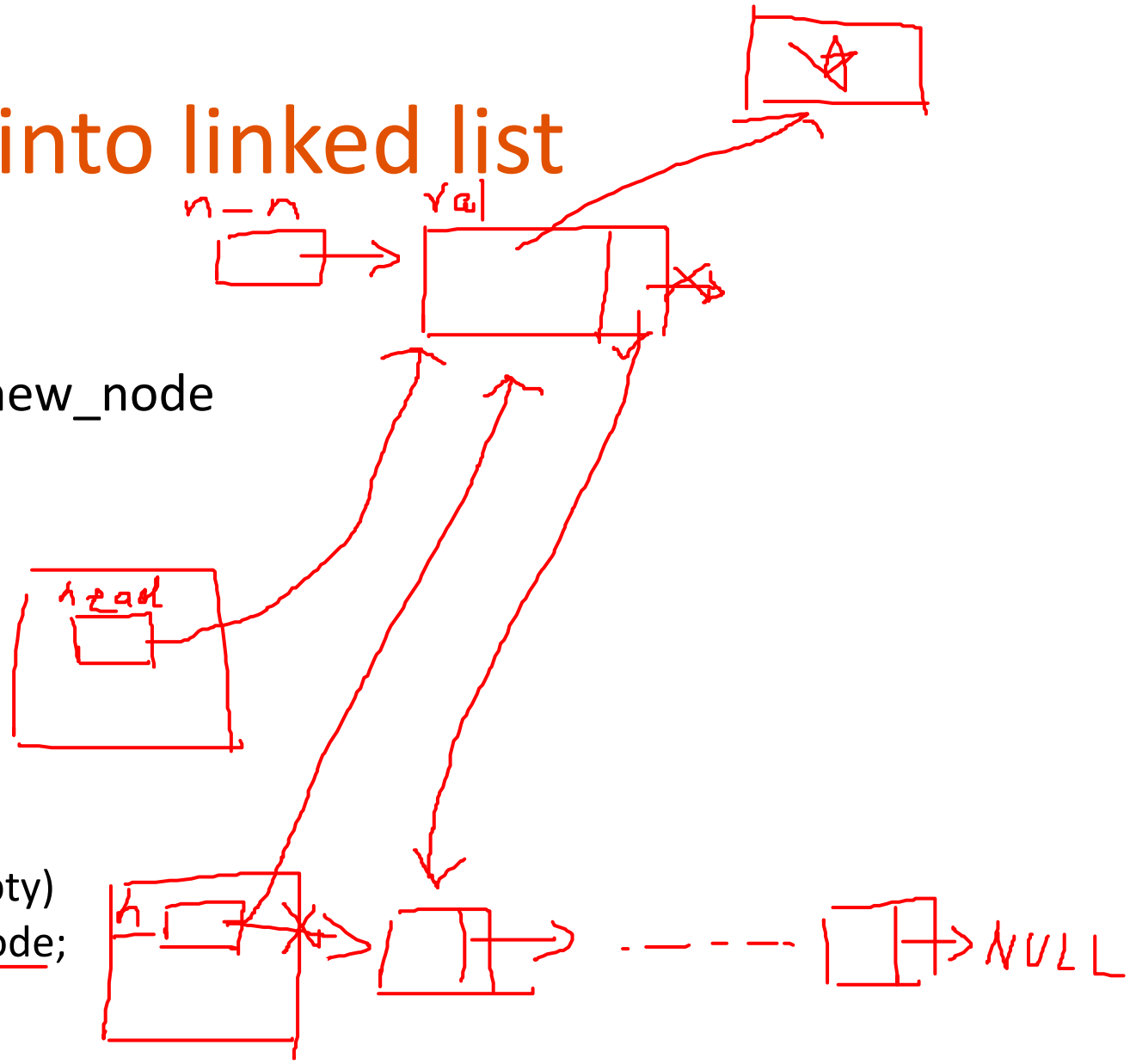
- Where can we insert?
  - Front/head
  - End/tail
  - Middle

Inserting an element into linked list

- Insert an element to the front:
  - Construct a node to be inserted, new_node
  - Assign new_node's next to NULL

  - Case 1:
    - Head is NULL (the list is empty)
    - Simply let head point to new_node

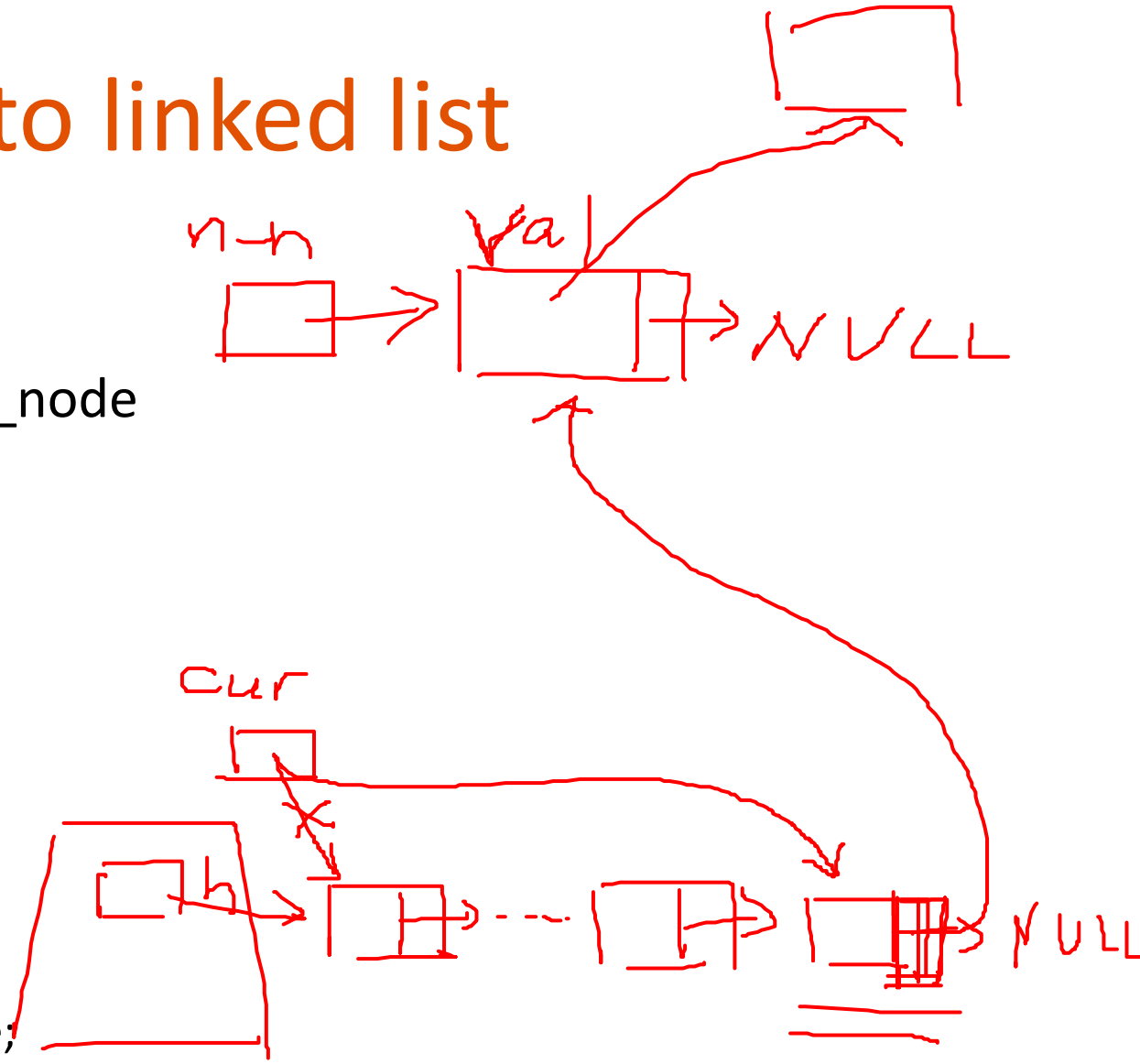
  - Case 2:
    - Head is not NULL (the list is not empty)
    - new_node's next points to the 1st node;
    - head point to new_node



Inserting an element into linked list

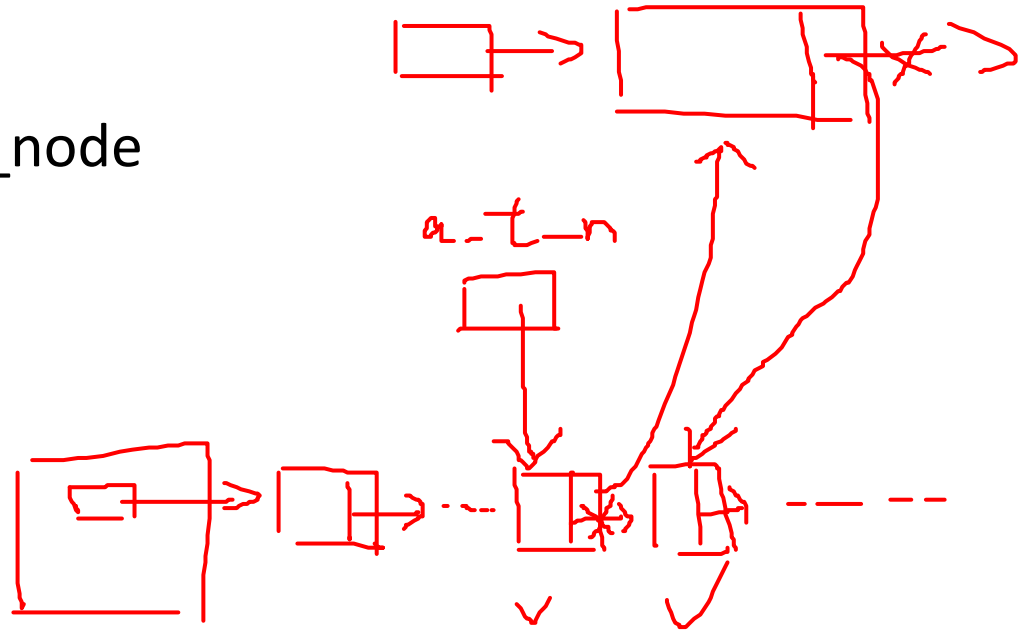
- Insert an element to the end:
  - Construct a node to be inserted, new_node

    n->next = NULL
  - Case 1:
    - Head is NULL (the list is empty)
    - Simply let head point to new_node
  - Case 2:
    - Head is not NULL (the list is not empty)
    - Loop to find the last element, last_node
    - last_node's next points to the new_node;



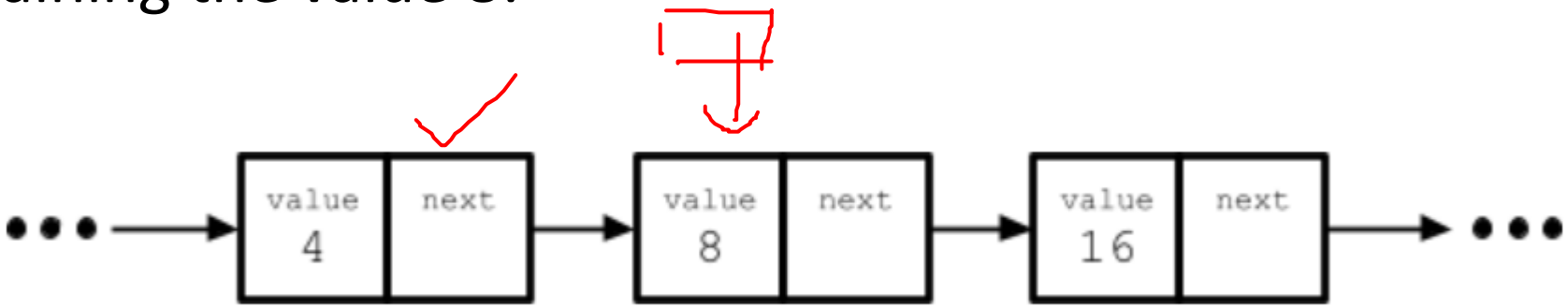
Inserting an element into linked list

- Insert an element to the middle:
  - Construct a node to be inserted, new_node
    n_n->next =NULL
  - Case 1:
    - Head is NULL (the list is empty)
    - Simply let head point to new_node
  - Case 2:
    - Head is not NULL (the list is not empty)
    - Loop to find the position to insert, after_this_node
    - new_node's next points to the after_this_node's next
    - after_this_node's next points to the new_node



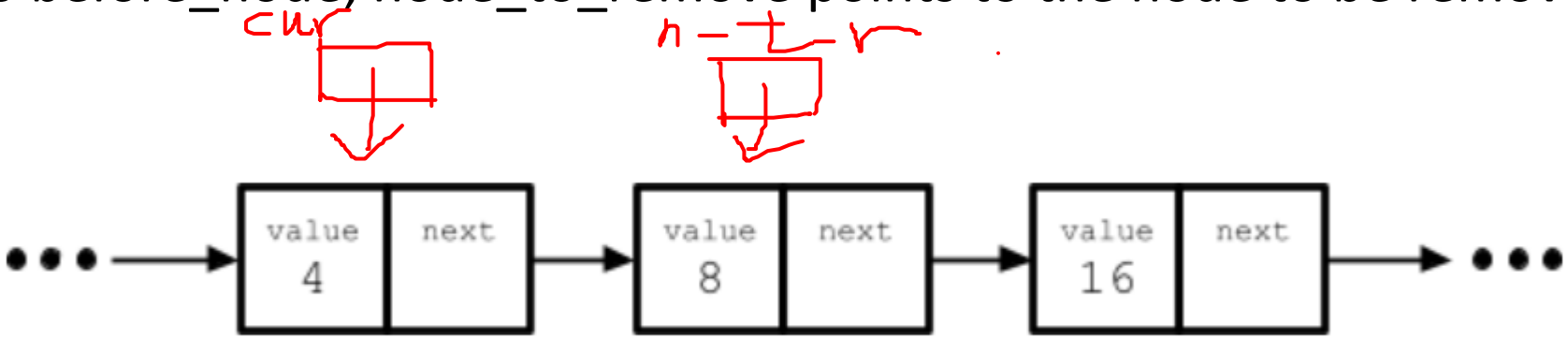
Removing and element from a linked list

- Opposite steps as inserting a new one
- Ex. Assuming the list is not empty, and we want to remove the node containing the value 8:



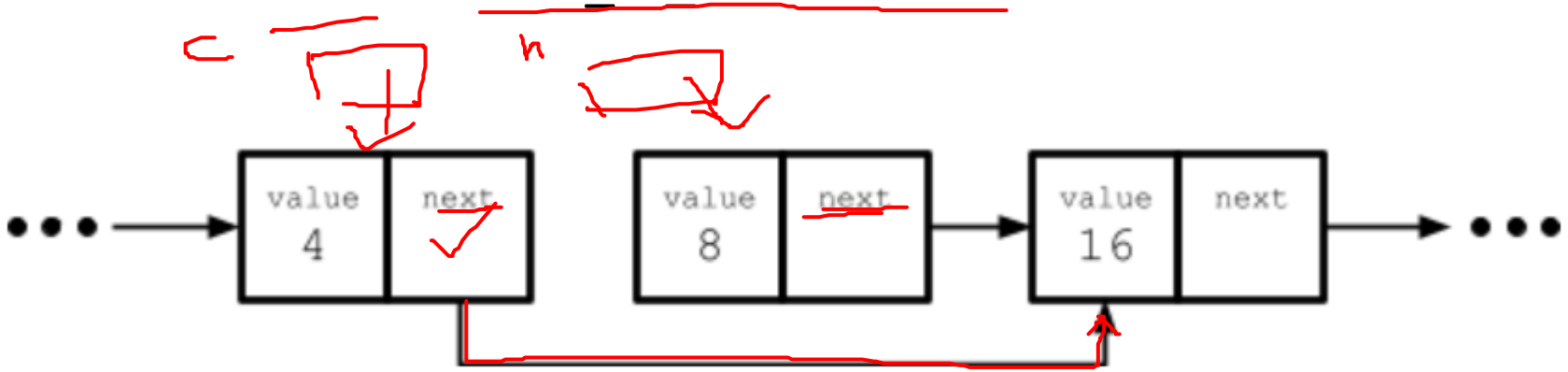
Removing and element from a linked list

- Step 1:
  - Loop to find the node to be removed and the node before, i.e., current points to before_node, node_to_remove points to the node to be removed



Removing and element from a linked list

- Step 2:
  - Set current's next to node_to_remove's next



Removing and element from a linked list

- Step 3:
  - free node_to_remove

Missing topic: Command-line Arguments

- Arguments passed into main()
  - `int main(int argc, char **argv)` OR
  - `int main(int argc, char *argv[])`

- Allow you to take input(s) from the user before running your program

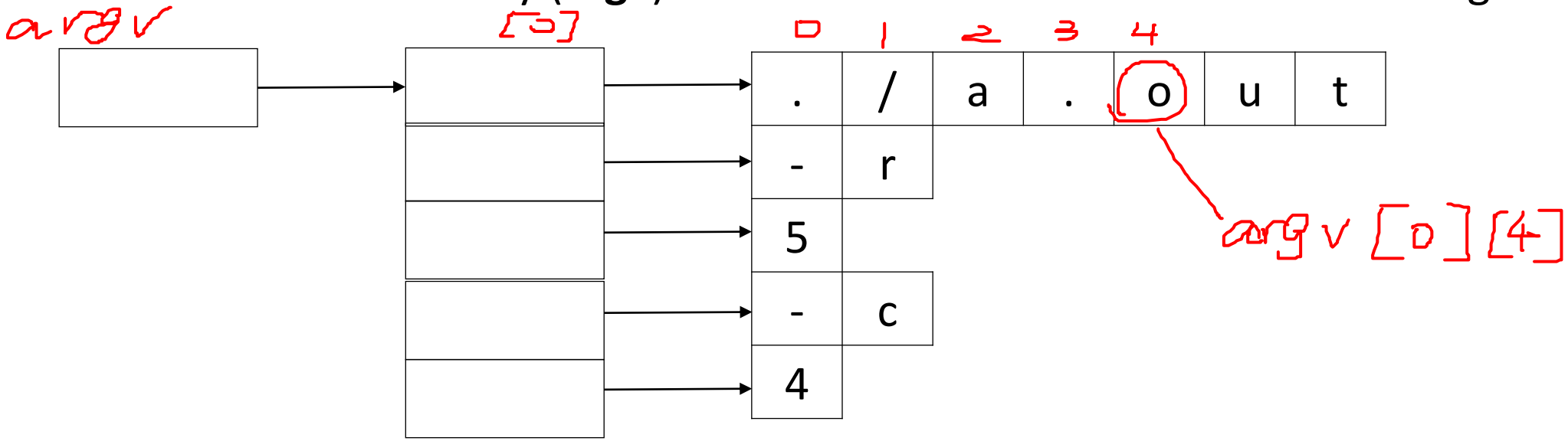
- argc: number of arguments
- argv: Array of c-style strings

Example:

- What is the value of **argc** if user entered this command to run the program?

```
./a.out -r 5 -c 4
```

- What does the 2-d array (**argv**) look like for the above command-line arguments?



Lecture Topics:

- Dynamic Array (cont.)
- Linked List
- Begin Complexity Analysis

How to compare Data Structures?

- We have different data structures, how to compare them?

- We want a way to characterize runtime or memory usage that is completely **platform-independent**
  - i.e. does not depend on hardware, operating system, programming language, etc.

Complexity Analysis

- Use Complexity Analysis to help make platform-independent comparisons of data structures
  - Also known as Big O

- To do this, we describe how a data structure's runtime or memory usage changes relative to a change in the input size (**n**)
  - Importantly, we want to describe how data structures behave in the limit, as n approaches ∞ (infinity)

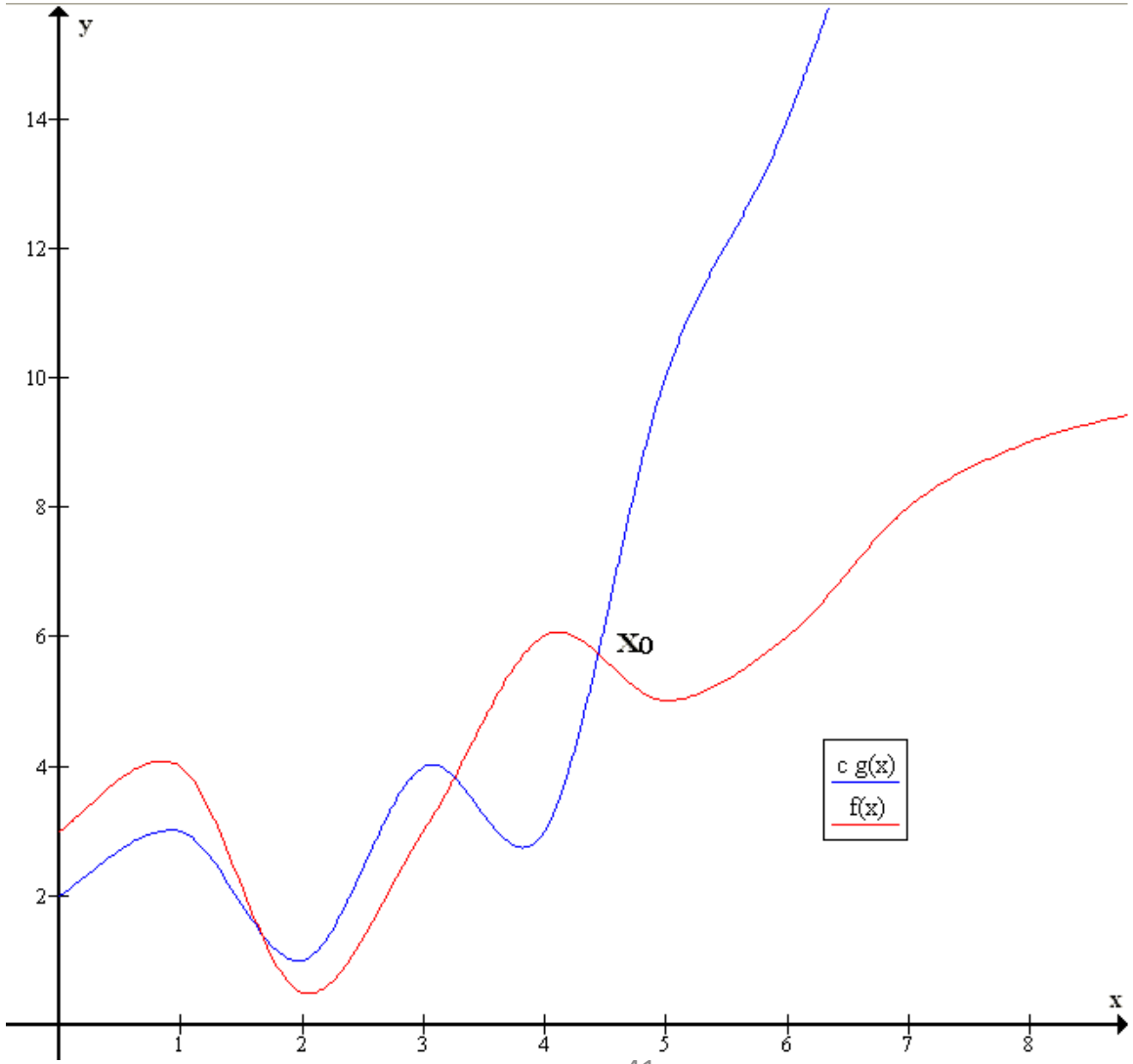
Big O

- We use Big O notation to assess a data structure or algorithm's performance.

- Big O notation: a tool for characterizing a function in terms of its growth rate
  - Indicate an upper bound on the function's growth rate, known as growth order

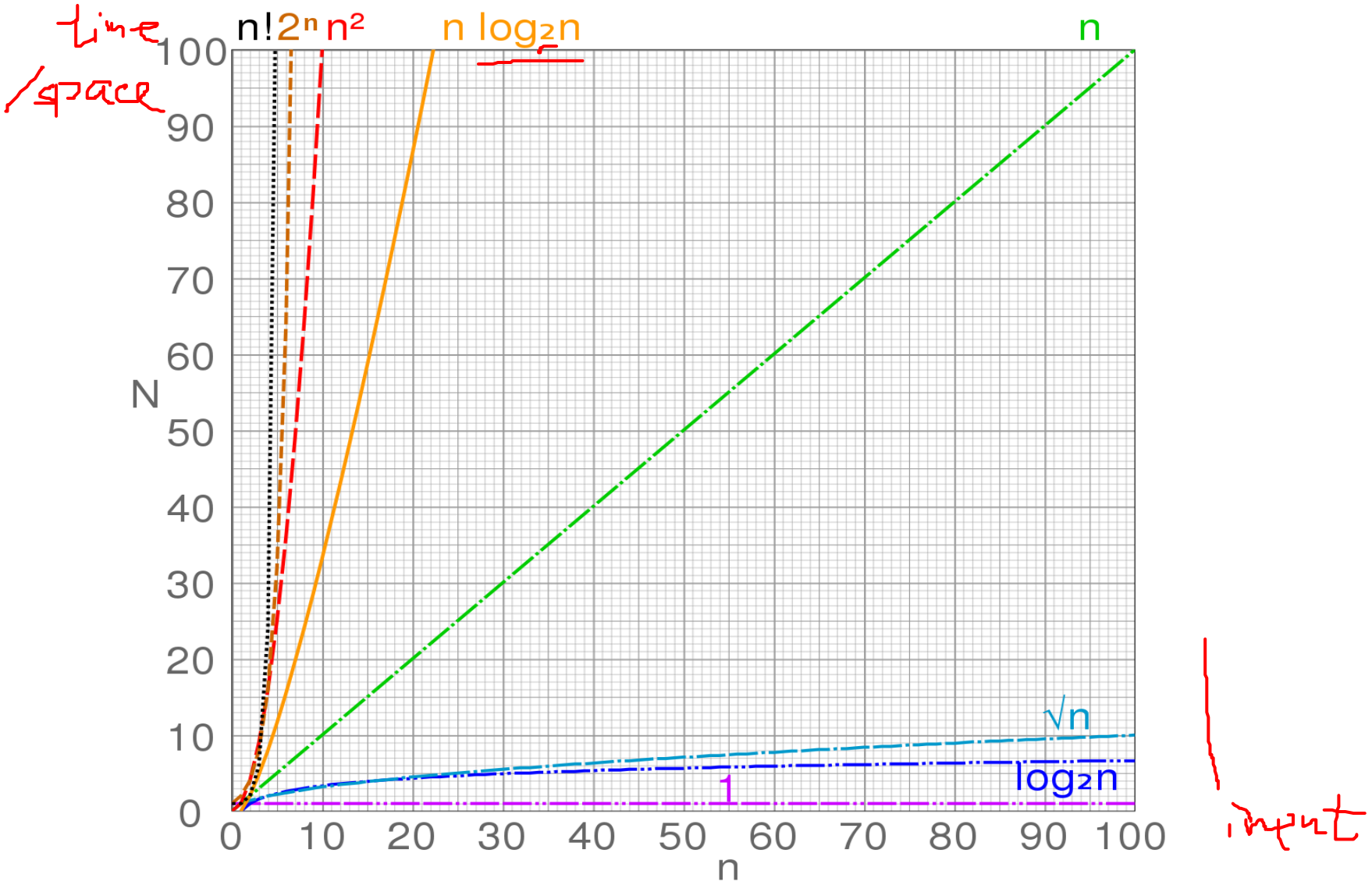
Big O

g(x) provides an upper bound on *f(x)*



g(x) is *O(f(x))*

Common growth order functions



Common growth order functions

- O(1) – constant complexity
- O(log n) – log-n complexity
- O(√n) – root-n complexity
- O(n) – linear complexity
- O(n log n) – n-log-n complexity
- O(n^2) – quadratic complexity
- O(n^3) – cubic complexity
- O(2^n) – exponential complexity
- O(n!) – factorial complexity