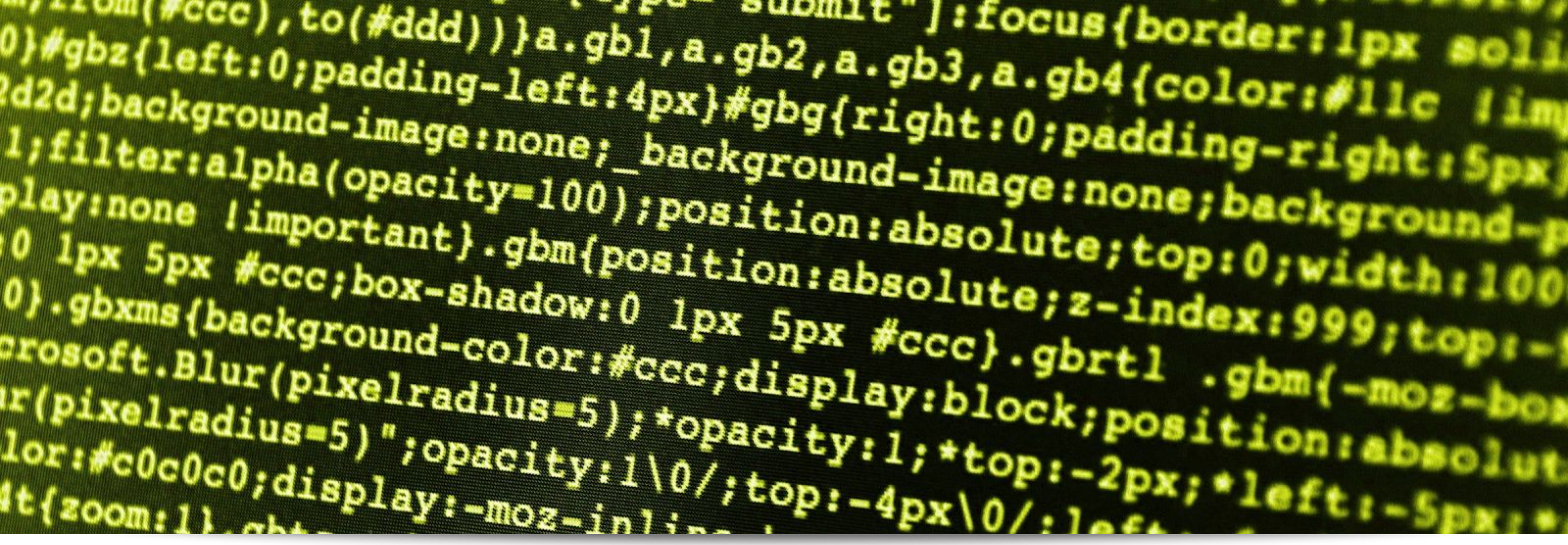


02

January 23-29, 2024

Arrays, Linked Lists, Stacks, Queues, Deques

Announcements

- Project 1 is due on **Thursday 2/01** (11:59 PM EDT).
  - If you submit on **Friday**, **2/02**, you must use ONE late day.
  - If you submit on **Saturday**, **2/03**, you must use your SECOND late day - even if you DID NOT submit on Friday, you would have to use both late days.
- Lab 1 autograder and quiz are due on **Monday, 1/29**.
- Lab 2 written problem is due on **Monday, 1/29, IN LAB**
- Lab 2 autograder and quiz are due on **Friday, 2/05**.

Agenda

- Intro to Complexity Analysis
- Arrays
- Linked Lists
- Stacks
- Queues
- Deques

Complexity Analysis

Complexity Analysis (Recall from 203)

- The concept of asymptotic runtime: how does the runtime of an algorithm scale as I increase the size of the input? If I double the input, does the runtime...
  - stay relatively constant? $O(1)$
  - also double? $O(n)$
  - quadruple? $O(n^2)$
- When looking at asymptotic runtime, we can eliminate coefficients and lower order terms, e.g. $O(5n^2 + 16n + 47)$ is simply $O(n^2)$.
- Asymptotic runtime only tells us how runtime scales with input size, *not* the actual runtime of an algorithm!

Complexity Analysis

```
Which algorithm has a better time complexity?
Which algorithm is (usually) going to be faster?

void foo(int n) { //O(1), usually slower
  for (int i = 0; i < 100000000; ++i) {
     cout << n << endl;
  }
}

void bar(int n) { //O(n), usually faster
  for (int i = 0; i < n; ++i) {
     cout << n << endl;
  }
}
```

Complexity Analysis

- Consider the complexity of the following function:

  $f(n) = 3n^3 - 3n^2 + 4n + 2$

- Which of the following statements are true?
  A) $f(n) = O(3n^2)$
  B) $f(n) = O(3n^3 - 3n^2)$
  C) $f(n) = O(2n^3)$
  D) $f(n) = O(n^3)$
  E) $f(n) = O(n^n)$

Complexity Analysis

- For which of the following pairs of f(n) and g(n) is f(n) = O(g(n))?

A)	f(n) = 1 + n/6	g(n) = √n + 5 log n + 7	O(n) vs. O(√n)
B)	f(n) = 3^{3^n}	g(n) = 3^{3n}	O(3^{3^n}) vs. O(3^{3n})
C)	f(n) = $4n^2$	g(n) = $2n^2$ - 10n	Both O(n^2)
D)	f(n) = ln(n)	g(n) = ln(n/10)	Both O(ln(n))
E)	f(n) = log($n^2 3^n$)	g(n) = 6n + 9	Both O(n)

Complexity Analysis

- What is the time complexity of the following code?
  - Hint: binary search is a $\Theta(\log m)$ process, where m is the number of columns our 2D array.

```
// Accepts an n x m array and an item to search for
// Assumes that each individual row is already sorted
void array2Dsearch(const vector<vector<int>> &array2D, int item) {
  for (size_t i = 0; i < array2D.size(); ++i) {
    if (binary_search(array2D[i].begin(), array2D[i].end(), item)) {
        cout << "found " << item << '\n';
        return;
    } // if
  } // for
  cout << "did not find " << item << '\n';
} // Total complexity (tightest bound): ???
```

We are doing a binary search on all columns of a row, where m is the number of columns. Thus, this process takes $\Theta(\log m)$ time.

Complexity Analysis

- What is the time complexity of the following code?
  - Hint: binary search is a Θ(log m) process, where m is the number of columns our 2D array.

```
// Accepts an n x m array and an item to search for
// Assumes that each individual row is already sorted
void array2Dsearch(const vector<vector<int>> &array2D, int item) {
  for (size_t i = 0; i < array2D.size(); ++i) {
    if (binary_search(array2D[i].begin(), array2D[i].end(), item)) {
        cout << "found " << item << '\n';
        return;
    } // if
  } // for
  cout << "did not find " << item << '\n';
} // Total complexity (tightest bound): ???
```

This Θ(log m) binary search is done many times in the for loop. How many times? The number of rows in the 2D array, or n times.

Complexity Analysis

- What is the time complexity of the following code?
  - Hint: binary search is a Θ(log m) process, where m is the number of columns our 2D array.

```
// Accepts an n x m array and an item to search for
// Assumes that each individual row is already sorted
void array2Dsearch(const vector<vector<int>> &array2D, int item) {
  for (size_t i = 0; i < array2D.size(); ++i) {
    if (binary_search(array2D[i].begin(), array2D[i].end(), item)) {
        cout << "found " << item << '\n';
        return;
    } // if
  } // for
  cout << "did not find " << item << '\n';
} // Total complexity (tightest bound): Θ(n log m)
```

Thus, we are doing a Θ(log m) process n times, for a total complexity of Θ(n log m)

Complexity Analysis

- If you have an algorithm with two steps, when do you multiply the complexities, and when do you add them?
  - If your algorithm is in the form "do this, then when you are done, do that" - you add the runtime complexities.
  - In this example, we do x work and then y work, so the complexity is $\Theta(x + y)$.

```
for (int x : vecX) {
   cout << "constant time work\n";
}
for (int y : vecY) {
   cout << "more constant time work\n";
}
```

Complexity Analysis

- If you have an algorithm with two steps, when do you multiply the complexities, and when do you add them?
  - If your algorithm is in the form "do this for each time you do that" - you multiply the runtime complexities.
  - In this example, we do y work each time we do x work, so the complexity is Θ(xy).
  - Nested for loops are often a key indication that you are multiplying complexities.

```
for (int x : vecX) {
   for (int y : vecY) {
      cout << "constant time work\n";
   }
}
```

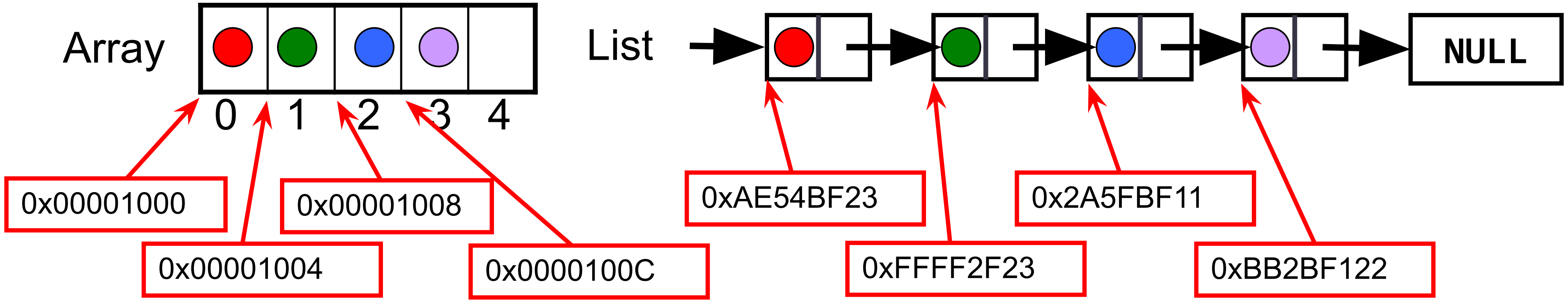
Complexity Analysis

- Sometimes, it may be difficult to identify the complexity of algorithm through simple observation.
- We have tools that can be used when recurrence is involved:
  - substitution method
  - Master Theorem
- We'll cover these at the beginning of the next lab!

Arrays and Linked Lists

Arrays vs. Linked Lists

- Two notable methods to represent (ordered) lists (an ADT) in memory
- Memory Layout
  - Arrays are allocated in a single contiguous chunk in memory
  - Linked lists are allocated in non-contiguous chunks in memory
- Pointer arithmetic only works with arrays, so distance between elements can be found in $\Theta(1)$ time for arrays, but only $\Theta(n)$ for lists



Arrays vs. Linked Lists: A Comparison

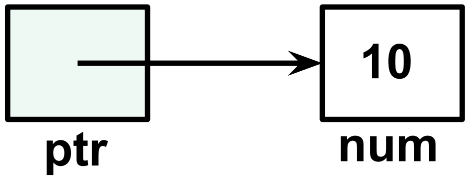
	Arrays	Linked Lists
Access	Random in O(1) time Sequential in O(1) time	Random in O(n) time Sequential in O(1) time
Insert and Append	Inserts in O(n) time Appends in O(n) time (O(1) amortized possible if vector)	Inserts in O(n) time Appends in O(n) time (O(1) with tail ptr)
Bookkeeping	Ptr to beginning CurrentSize or ptr to end of space used (optional) MaxSize or ptr to end of allocated space (optional)	Size (optional) Head ptr to first node Tail ptr to last node (optional) In each node, ptr to next node
Memory	Wastes memory if size is too large* Requires reallocation if too small*	Allocates memory as needed Memory overhead for pointers (wasteful for small data items)

*With C++ vectors, these issues can be alleviated with the reserve() or resize() methods if you know what the size is going to be upfront

Arrays and Pointers

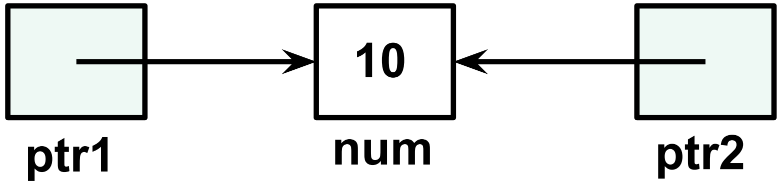
- Pointers store address locations

```
int num = 10;
int* ptr = &num;
```

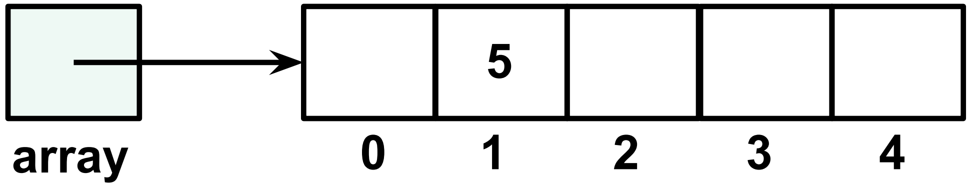


- Multiple pointers to one address location

```
int num = 10;
int* ptr1 = &num;
int* ptr2 = ptr1;
```



- Arrays == Pointers



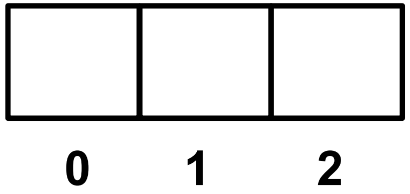

```
int* array = new int[5];
array[1] = 5;
cout << array[1] << " " << *(array+1); // what does this output?
```

Array Resizing (Recall from 280)

- Strings, vectors, and other “auto-resizing” containers are implemented with arrays, reallocating their array and moving their data when needed.
    - This invalidated pointers
    - Only an issue if vector’s capacity changes
    - <u>Use resize and reserve when possible</u>
- Linked lists don’t have this problem!

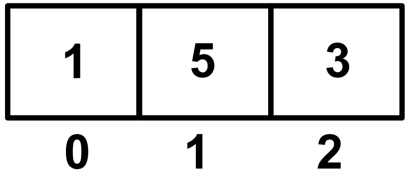
Array Resizing Example

```
vector<int> v;
```



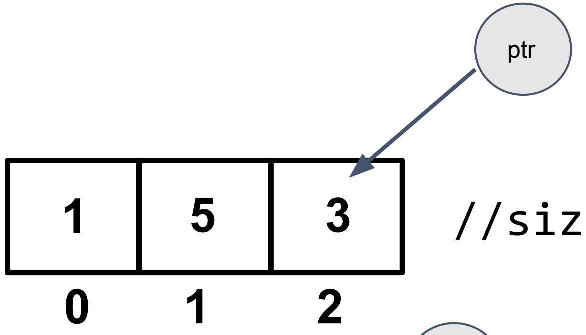

```
//size = 0, capacity = 3
(pretend C++ vectors have a
default capacity of 3)
```

```
v.push_back(1);
v.push_back(5);
v.push_back(3);
```



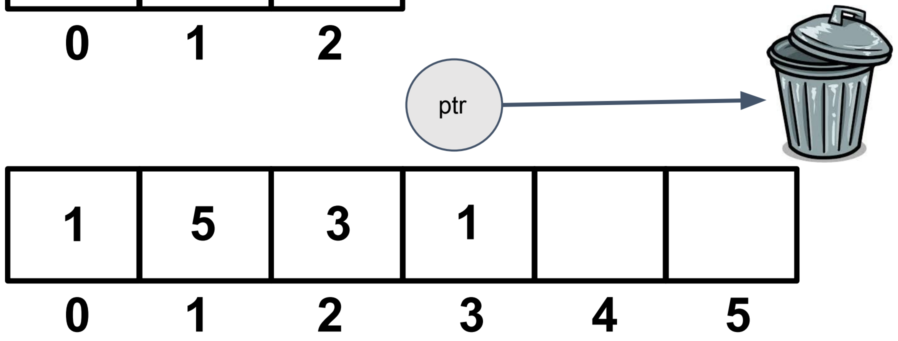

```
//size = 3, capacity = 3
```

```
int * ptr = &v[2];
```




```
//size = 3, capacity = 3
```

```
v.push_back(1);
```




```
//size = 4, capacity = 6
```

Practice Questions

- Which one is better at each of the following?
  - Given a pointer to an element, insert a new element right after?
  - Given an index, update an arbitrary element?
  - Search on a sorted container?
  - Remove multiple elements from a container?

Practice Questions

- Which one is better at each of the following?
  - Given a pointer to an element, insert a new element right after?

Linked list - no potential shifting of elements after insertion point

  - Given an index, update an arbitrary element?
  - Search on a sorted container?
  - Remove multiple elements from a container?

Practice Questions

- Which one is better at each of the following?
  - Given a pointer to an element, insert a new element right after?

Linked list - no potential shifting of elements after insertion point

  - Given an index, update an arbitrary element?

Array - random access to element, you have to Θ(n) search in a list

  - Search on a sorted container?
  - Remove multiple elements from a container?

Practice Questions

- Which one is better at each of the following?
  - Given a pointer to an element, insert a new element right after?

Linked list - no potential shifting of elements after insertion point

  - Given an index, update an arbitrary element?

Array - random access to element, you have to Θ(n) search in a list

  - Search on a sorted container?

Array - you can do binary search on an array, but not a linked list

  - Remove multiple elements from a container?

Practice Questions

- Which one is better at each of the following?
  - Given a pointer to an element, insert a new element right after?

Linked list - no potential shifting of elements after insertion point

  - Given an index, update an arbitrary element?

Array - random access to element, you have to $\Theta(n)$ search in a list

  - Search on a sorted container?

Array - you can do binary search on an array, but not a linked list

  - Remove multiple elements from a container?

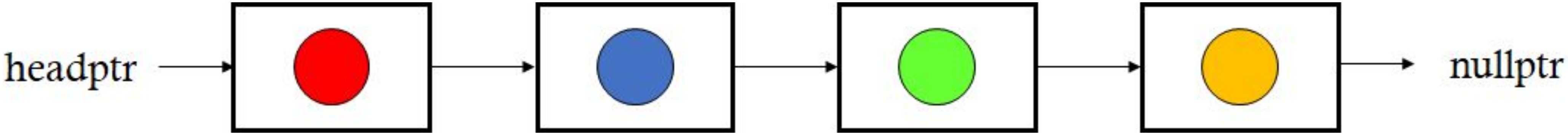
Linked list - no potential shifting of elements after deletion point

Linked Lists: The Two Pointer Technique

- The two pointer technique is a technique that you can use if you are ever asked a linked list question during an interview.
- Iterate through the list with two pointers simultaneously, with one either a fixed distance from the other, or one that moves faster than the other (slow and fast).

Linked Lists: The Two Pointer Technique

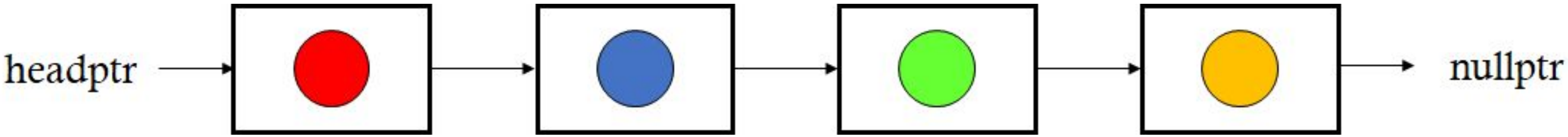
- **Example 1:** Given an integer *k*, find the k^{th} to last element in a singly-linked list. You do not know the length of the linked list.



- How can you solve the problem? *Use two pointers!*
  - The k^{th} to last element is *k* from the end of the list.
  - We can take two pointers that are a distance of *k* nodes apart, `fast` and `slow`. We start from the beginning and increment both until `fast` reaches the end of the list. Since `slow` is *k* nodes behind `fast`, `slow` must point to the k^{th} to last element!
  - O(*n*) time and O(1) space

Linked Lists: The Two Pointer Technique

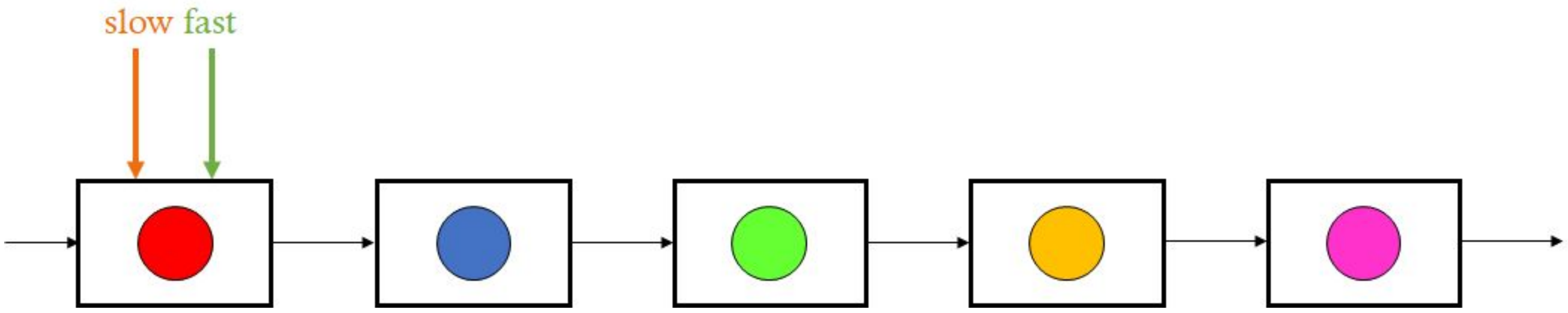
- **Example 2:** Given a singly-linked list, devise an algorithm that returns the value of the middle node. If there are two middle nodes, return the value of the second middle node.



- How can you solve the problem? *Use two pointers!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!

Linked Lists: The Two Pointer Technique

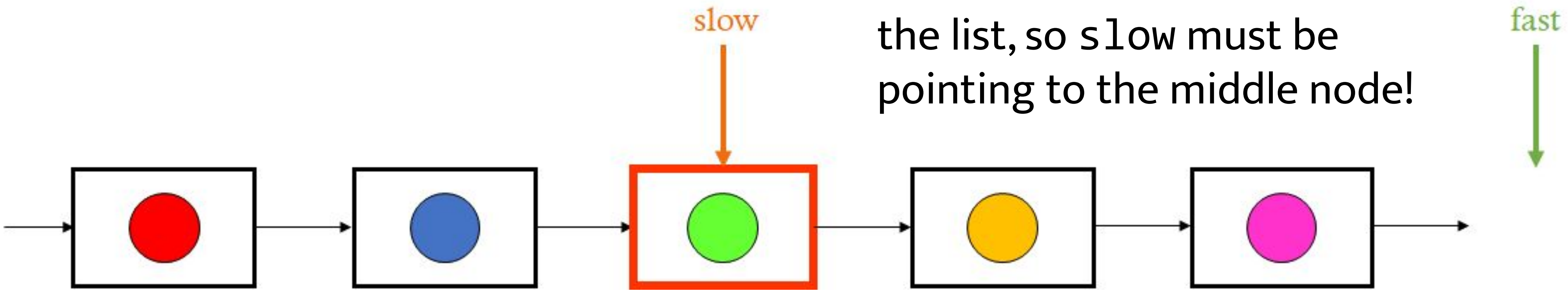
- How can you solve the problem? *Use two pointers!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!



Linked Lists: The Two Pointer Technique

- How can you solve the problem? *Use two pointers!*
  - Start with two pointers, `fast` and `slow`.
  - Increment `fast` by two, then increment `slow` by one.
  - When `fast` reaches the end, `slow` must point to the middle node!

`fast` has reached the end of the list, so `slow` must be pointing to the middle node!



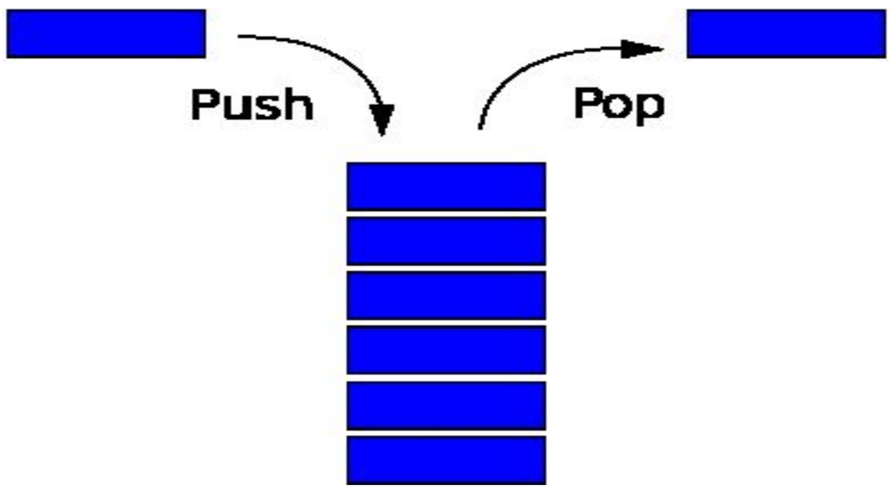
Stacks and Queues

Stacks and Queues

- Containers for “pushing” and “popping”
  - push();
  - pop();
  - top(); *(for stacks)*
  - front(); *(for queues)*
  - size();
  - empty();
- NO RANDOM ACCESS!

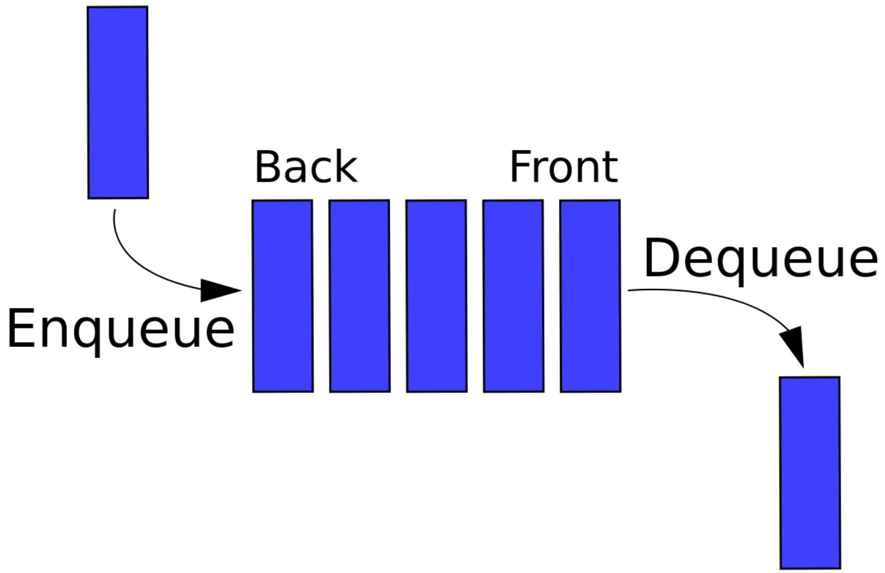
Stacks

- Last-in, First-out (LIFO)
- Member functions
  - push();
  - pop();
  - top();
  - size();
  - empty();
- Real life examples?



Queues

- First-in, First-out (FIFO)
- Member functions
  - push();
  - pop();
  - front();
  - size();
  - empty();
- Real life examples?



Interview Question: Sorting a Stack

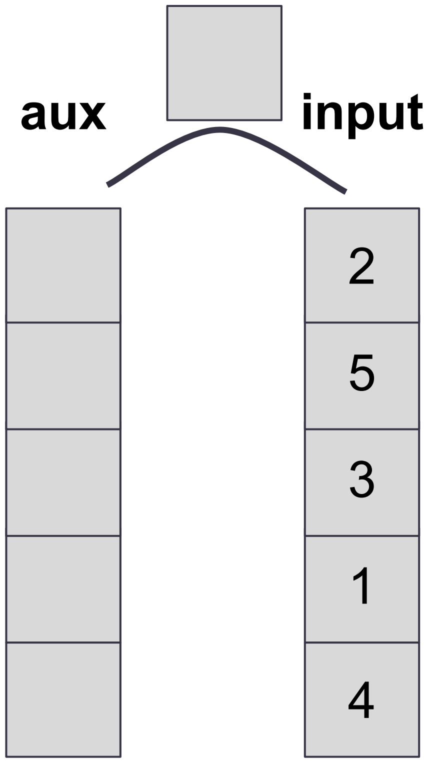
- Can you sort a stack using only an auxiliary stack and O(1) additional space? This stack supports top(), push(x), pop(), and size().
- How should this problem be approached?
  - **Let's make use of our auxiliary stack (aux)! Even though our input stack is not sorted, we can push elements into this auxiliary stack in such a way to keep aux sorted.**
- Invariant: aux is sorted
- Invariant: no items are lost (they're either in aux or in the input stack after each iteration)

Interview Question: Sorting a Stack

- Push elements from the input stack into the auxiliary stack such that **large items** are sent to the back of the auxiliary stack.
    - Save the top of the input stack somewhere (we'll call this the **current element**)
    - Since we want larger items to be sent to the auxiliary stack, we'll return all elements in aux that are smaller than the current element back to the input stack
    - Once the auxiliary stack only contains elements larger than the current element, we can push this current element into this auxiliary stack

```
repeat until input.empty():
    x = input.top(); input.pop()
    while (!aux.empty() && aux.top() < x):
        input.push(aux.top()); aux.pop()
    aux.push(x)
```

Interview Question: Sorting a Stack



Order: 1 2 3 4 5

Let's push elements from the input stack into the auxiliary stack!

- Keep larger elements at the "bottom" of aux.
- Compare each element you want to add to aux with the top of aux - if the current element is larger, move stuff out of aux so that you can put the current element in the correct position.

See full lab slides for animation

Question: what is the worst-case runtime using this algorithm?

Interview Question: Implement Queue with Stacks

- Can you implement a queue using two stacks?
- The stacks support top(), push(x), pop(), and size().
- Your queue must support:
  - insert(x)
  - front()
  - remove()
  - size()

Interview Question: Implement Queue with Stacks

- Here's an idea: one of the stacks is the "front" (where elements are removed) and the other stack is the "back" (where elements are inserted).
- Is there a problem with this implementation?
- What if the front stack becomes empty? .top() and .pop() won't work!

```
stackFront and stackBack:
  insert(x): stackBack.push(x)
  front(): stackFront.top()
  remove(): stackFront.pop()
  size(): stackFront.size() + stackBack.size()
```

Solution: fix the queue whenever this happens!

Interview Question: Implement Queue with Stacks

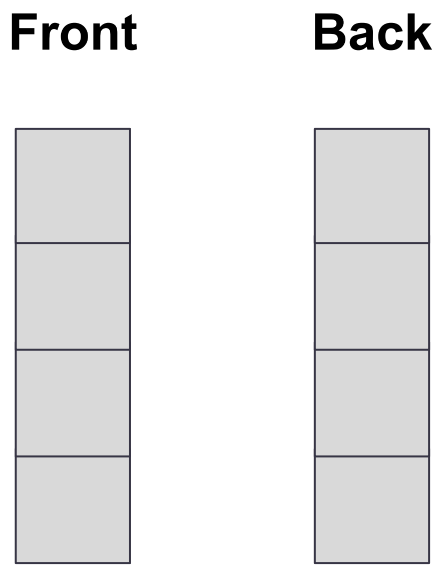
- If `stackFront` ever becomes empty, and a front() or remove() operation is invoked, move elements from `stackBack` into `stackFront`!

```
remove():
    if (stackFront.empty()) {
        while (!stackBack.empty()) {
            stackFront.push(stackBack.top());
            stackBack.pop();
        }
    }
    stackFront.pop();
```

Question: why not move just one element?
Answer: the element we need to remove is at the bottom of `stackBack`, not the top!

Interview Question: Implement Queue with Stacks

See full lab slides for animation

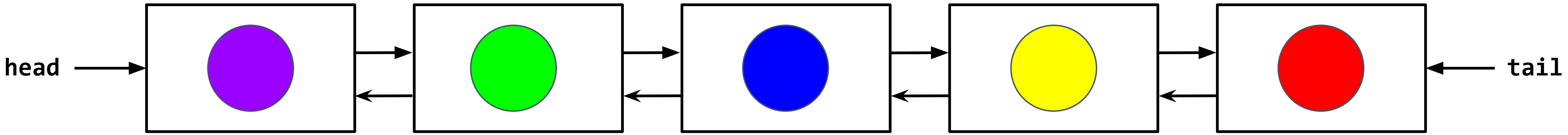


insert()
insert()
insert()
pop()
insert()
insert()
pop()
pop()
pop()

Deques

Deques

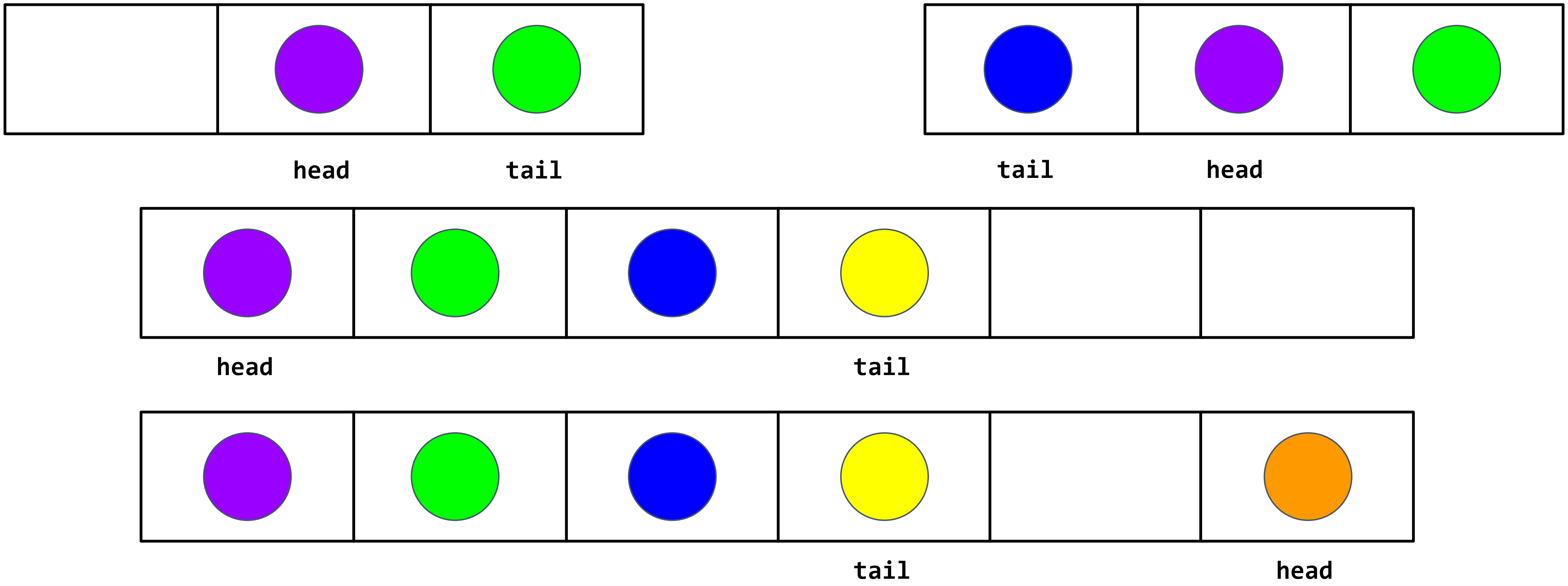
- A deque can be used to support efficient front and back access!
  - a stack and queue all in one
  - but can also traverse using iterators and supports `operator[]` access
  - Efficiently implements the list ADT, but with more functionality than vectors
- Supports O(1) `.push_front()`, `.push_back()`, `.pop_front()`, `.pop_back()`, `.front()`, `.back()`, and `operator[]`.
- A simple implementation: a doubly-linked list:



- Potential problems with this implementation? operator[] isn't O(1)!

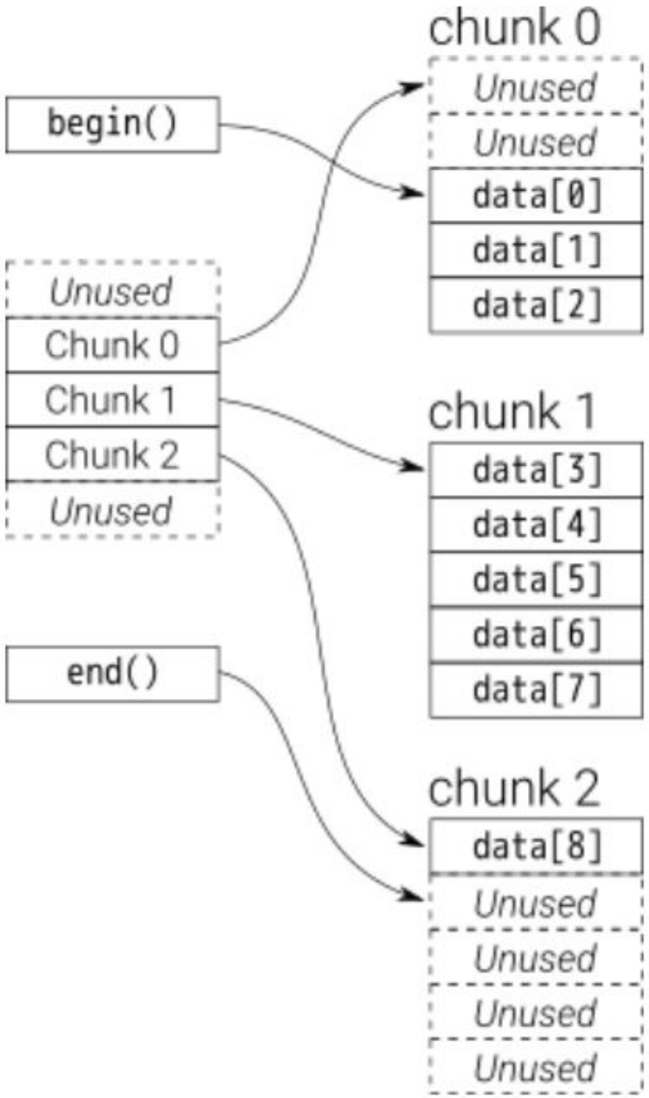
Deques

- Another possible implementation: a circular array:



- Potential problems with this implementation? Pointer invalidation

Deques



- The STL deque is essentially a deque of deques
  - dynamic array of pointers to dynamic arrays of a fixed size (the "chunk size") , which are allocated as necessary
  - data can be pushed and popped from both ends as needed
  - pointers remain valid upon reallocation of outer array, since they lie in the chunks, and allocation of new chunks does not affect older chunks. This is useful because indices cannot replace this.
- With some math, this supports O(1) `operator[]`
  - suppose we want to retrieve the element at index 7
    - we add 7 to the number of unused slots in chunk 0: **7 + 2 = 9**
    - dividing this by the chunk size gives us the chunk the element is in: **9 / 5 = 1** (integer division truncates all decimals in C++)
    - taking the modulo gives us the position of the element within its chunk: **9 % 5 = 4**
  - thus, element 7 of this deque is at index 4 of chunk 1

Handwritten Problem

Handwritten Problem

- Implement a queue with a singly linked list:

```
template <typename T>
class LinkedQueue {
private:
    Node<T>* head  = nullptr;
    Node<T>* tail  = nullptr;
    size_t   count = 0;
public:
    T      front() const { /* Implement */ }
    void   pop()         { /* Implement */ }
    void   push(T x)     { /* Implement */ }
    size_t size() const  { return count; }
    bool   empty() const { return count == 0; }
     ~LinkedQueue()      { /* Implement */ }
};
```

```
template <typename T>
struct Node {
    T     value;
    Node* next;
};
```