Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Lists I: Arrays, Dynamic Lists, and Linked Lists

**"What has a head, a tail and no body?
Answer: A Linked List"**

CS2013: Programming with Data Structures

Data Structures

- ***data structure***: a particular way of organizing data in a computer so that it can be used efficiently. Also supports operations for accessing and manipulating data.
  - also known as a ***container*** or ***container object***
  - the items it stores are ***data*** or ***elements***.

- In an object oriented language, data structures are usually implemented as a class.
  - data fields store the data / elements
  - methods provide additional capabilities for searching, insertion, deletion, and other operations.

Abstract Data Types (ADTs)

- ***abstract data type***: a model for data types, where a data type is defined by its behaviors from the point of view of a ***user*** of the data.

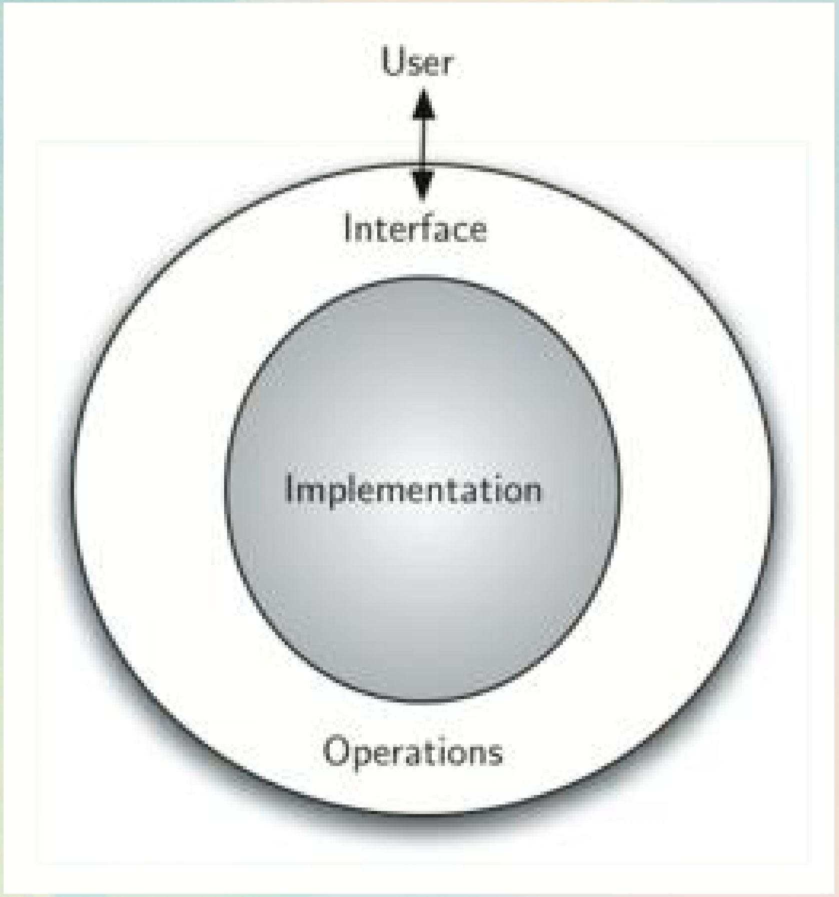
- This is a logical description of how we view the data and the operations that are allowed with that data, with no regard for how those operations are implemented.

- NOTE: this has nothing to do with the concept of abstract classes, and more to do with the idea of ***abstraction*** where a data structure is defined using a very high level description, leaving out implementation details.

Abstract Data Types (ADTs)

- an ADT is also independent of any specific implementation.
  - Example: You can implement a List type data structure in Java using OOP, or you could implement the same type of data structure in a language like Haskell which is a functional programming language.

- This picture illustrates what we mean by ADT where the user interacts with the ***public interface*** of the data structure and has little to no knowledge of how that data structure is implemented.



Arrays

Arrays

- ***array***: a data structure consisting of a collection of elements, each identified by at least one *array index* or *key*.

- Arrays are stored so that each position of each element can be computed from its index by a simple mathematical computation.
  - memory is allocated in one huge block of sequential memory locations

- Arrays are some of the oldest and most important data structures, and are used by almost every computer program.

- Arrays are also used to implement many other data structures such as Dynamic Lists and Strings.

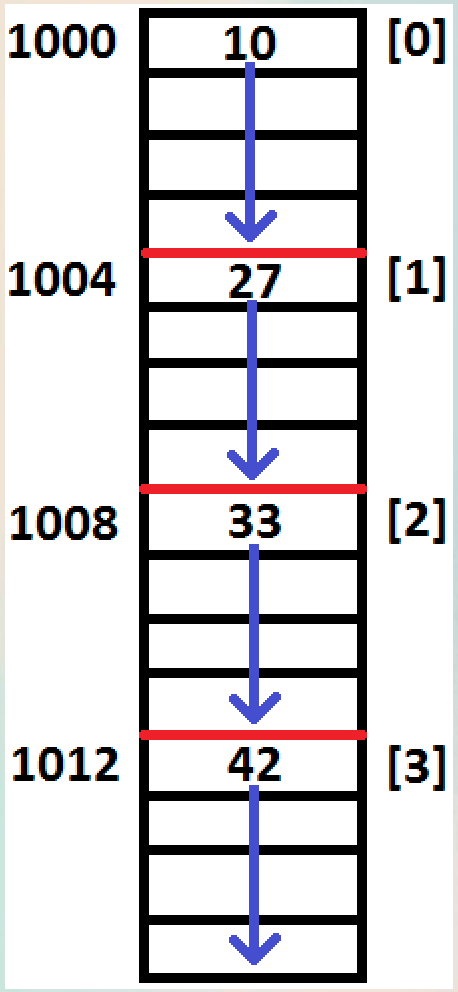
Array Big-O Analysis

- **Access: O(1)**
  - Arrays provide random access.
  - To access any element of the array, the address of an element is computed as an offset from the ***base address (foundation address***) of the array.
    - element address = base address + (index * data type size)
    - access requires one multiplication and one addition operation which both run in constant time.
- **Searching: O(n)**
  - You have to search through the entire array to find what you are looking for.
- **Insertion: O(n)**
  - You usually have to shift elements to the right in order to create an insertion point.
- **Deletion: O(n)**
  - You usually have to shift elements to the left in order to delete an element.
- **Space Complexity: O(n)**
  - Arrays have n elements and the size of the array determines how much space in memory the array takes up.

Example of Array Access

- Suppose:
  - you have a 4 element array of integers
  - each integer takes up 4 bytes of memory
  - *element address =*
    *base address + (index * data type size)*

- Computations for Element Access:
  - Index [0] Element: 1000 + (0 * 4) = 1000
  - Index [1] Element: 1000 + (1 * 4) = 1004
  - Index [2] Element: 1000 + (2 * 4) = 1008
  - Index [3] Element: 1000 + (3 * 4) = 1012



Arrays Pros and Cons

- Pros:
  - Simple and easy to use.
  - Usually a fundamental part of any programming language.
  - Constant time access of elements.
  - Can be used to implement many other data structures.
  - Random access.

- Cons:
  - ***static size***: the size is specified at array creation and is fixed.
  - **one block allocation**: a block of sequential / consecutive memory addresses must be available to store the complete array. may not always be possible to have enough memory to store the entire array.
  - **complex position-based insertion**: to insert an element, you may need to shift the existing elements to create an insertion point. adding to the beginning is expensive because you must shift all the elements in the array.

Dynamic Lists

Dynamic Lists

- ***dynamic list*:**
  - a random access variable-size list structure that allows elements to be added or removed
  - Also called a dynamic array, growable array, resizable array, dynamic table, or array list, because the size of the array can grow and shrink as needed.
- usually implemented using an array of fixed size:
  - when the array capacity is reached, the dynamic list will resize the array to add more space to add more elements.
  - resizing usually consists of creating a new fixed array of larger size and then copying the old elements to the new array.

Pros and Cons of Dynamic Lists

- Pros:
  - same pros as arrays.
  - can be re-sized as needed (dynamically)
  - provides additional functionality above what an array offers. (search functions, display operations, etc)
- Cons:
  - **one block allocation**: same as array (because this uses an array underneath)
  - **complex position-based insertion**: same as array (because this uses an array underneath)
  - takes extra time to resize the array when more positions in the array are needed.
  - can waste memory (capacity of underlying array may be larger than the actual size of the data needed)
  - generally requires more memory than an array since there is overhead in creating a new Class (all objects have some overhead).

Size vs Capacity

- ***size***: when we talk about the size of a data structure, we mean the actual number of elements currently stored in the data structure.

- **capacity:** when we talk about the capacity of a data structure, we mean the total number of elements that can potentially be stored in the data structure (before a resize operation needs to take place.)
  - capacity is always >= size.

- **Example:** A Dynamic List with capacity 10 and size 7 means that there are 7 values stored in the list, but it could hold up to 3 more (10 total) before the underlying array would need to be re-sized.

Dynamic List ADT

- The ADT **`DynamicList`** is a linear sequence of an arbitrary number of items of the same type, which includes the following behaviors:

`DynamicList()`	Constructor which initializes a `DynamicList`. The constructor may be overloaded to accept an initial capacity, or another `DynamicList` to create a copy of said list, or other parameters for defining a `DynamicList`.
`append(item)`	Adds the given item to the end of the list.
`contains(item)`	Returns true if the `DynamicList` contains the specified item, false otherwise.
`delete(index)`	Removes the item at the given index.
`find(item)`	Returns the index of the first occurrence of where the item can be found, -1 otherwise.
`get(index)`	Returns the item in the `DynamicList` at the specified index.
`insert(index, item)`	Inserts an a new item at the position specified by the index.
`isEmpty()`	Returns true if `DynamicList` is empty, false otherwise.
`replace(index item)`	Replaces the item already at the given index with the given new item.
`size()`	Returns the current number of elements in the `DynamicList`.

Dynamic List - size()

- What is the runtime of size()?
- How should size be implemented?
- Think very carefully!!!

DynamicList Class Overview

- `DynamicList` has two data fields:
  - `int size;` //specifying the current size of the list.
  - `[] data;` //an array of integers holding the data.

DynamicList - resize()

- `DynamicList` has at least one private method:
  - `private resize():`
    - creates a new array **double** the capacity of the current data array.
    - copy the elements from the old array, to the corresponding indexes in the new array.
    - assign the new array to the `this.data` reference variable.
    - resize will always be O(n) (you always have to copy n elements from one array to the other).

- NOTE: Private methods are never part of the ADT model because private methods are never seen by the outside user.

DynamicList - resize()

- Caution: Notice how we will usually always have leftover space at the end of our array. You may be tempted to change the resize algorithm to only increase the size of the data array by one or two. This is bad!

- By increasing the size of the array by a constant size of 1, you are creating an arithmetic progression
  - $1 + 2 + 3 + 4 + n = (n(n+1))/ 2$ which we know is n^2

- This means that your resize() method will be called with any code that checks if a resize is necessary and will cause most of the runtimes of your DynamicList methods to become n^2.

- Usually we double the size of the current array, as this makes for a better result and resize() does not have to happen as often.

Dynamic List - append()

- `append(item)`:
  - Adds an item to the end of the list.
  - Resize the list if necessary.

- Runtime:

– Worst Case:	O(n) (Due to resizing)
– Best Case:	O(1)
– **Amortized Analysis:**	O(1)

```
append(item):
    if size == capacity: resize()
    data[size] = item
    size++
```

Dynamic List - contains()

- `contains(item)`:
  - Returns true if the item is in the list, false otherwise.
- Runtime: O(n)
  - On average, you will need to search through n items in the list.

```
contains(item):
  for all e in data:
    if e == item: return true

  return false
```

Dynamic List - delete()

- `delete(index)`:
  - Removes the item at the given index.
  - Variations of this return the deleted item.

- Runtime:
  - Worst Case: O(n) (Due to shifting n elements)
  - Best Case: O(1) (Deleting from the end)
  - **Amortized Analysis:** O(n)

```
delete(index):
  check that index is in bounds

  if index != size-1:
    for i from index to size-2:
      data[i] = data[i+1]

  size--
```

Dynamic List - find()

- `find(item)`:
  - Returns the index of the first occurrence of where the item can be found, -1 otherwise.

- Runtime: O(n)
  - On average, you will need to search through n items in the list.

```
find(item):
  for i from 0 to size-1:
    if e == item: return i

  return -1
```

Dynamic List - get()

- `get(index)`:
  - Returns the item in the `DynamicList` at the specified index.

- Runtime: O(1)
  - Because we have an array backing our DynamicList, the look-up time is constant.

```
get(index):
  check that index is in bounds

  return data[index]
```

Dynamic List - insert()

- `insert(index, item)`:
  - Inserts an a new item at the position specified by the index.

- Runtime: O(n)
  - Due to shifting n elements.

```
insert(index, item):
  check that index is in bounds
  if size == capacity: resize()

  if index == size:
    append(item)
  else:
    for i size-1 to index:
      data[i+1] = data[i]
    data[index] = item

  size++
```

Dynamic List - replace()

- `replace(index, item)`:
  - Replaces the item already at the given index with the given new item.
- Runtime: O(1)
  - Because we have an array backing our DynamicList, the replace time is constant.

```
replace(index, item):
  check that index is in bounds

  data[index] = item
```

Dynamic List - isEmpty() and size()

- `isEmpty()`:
  - Returns true if the size is 0, false otherwise.

- `size()`:
  - Returns the size of the `DynamicList`.
  - How should size be implemented? This is very important to think about...

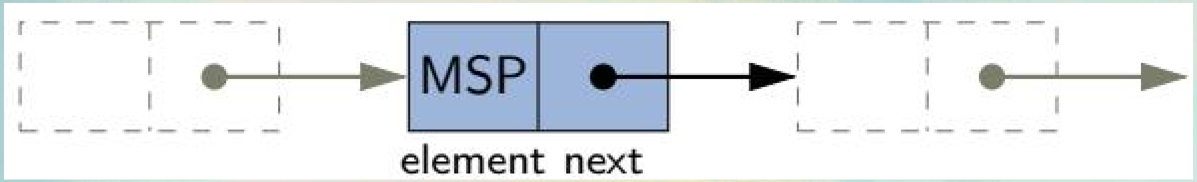
- Runtime: O(1) for both.
  - Each method only requires a constant amount of work.

Linked Lists

Linked Lists

- ***linked list***: a linear collection of data where the linear order is not determined by their physical placement in memory, but by how one piece of data links to another using pointers or references.

- a linked list is a collection of ***nodes*** that link together to form a linear sequence of data.



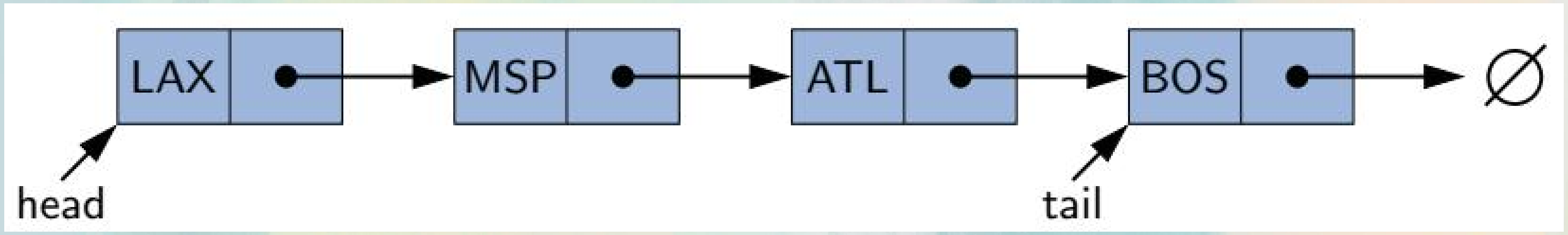
Linked List Terminology

- ***node***: the object which encapsulates the actual data of each element. Minimally, a node has a data field for the data of that element and a pointer to the next node in sequence.
- ***head***: the beginning of the linked list, usually refers to the first node in the list.
- ***tail***: the end of the linked list, usually refers to the last node in the list.
- ***head pointer (reference)***: the reference variable in the linked list class which points to the first node of the list.
- ***tail pointer (reference)***: the reference variable in the linked list class which points to the last node of the list.
- ***forward pointer (reference)***: the reference variable in a node which points to the next element in the list.
- ***backward pointer (reference):*** the reference variable in a node which points to the previous element in the list.
- ***pointer / reference / link hopping***: the process of traversing the linked list by starting at the head (or tail) and *hopping* from node to node by following the pointers.

Linked List Pros and Cons

- Pros:
  - can append elements to the list in constant time.
    - with arrays, the array might need to be resized which takes time,
    - with a linked list we just create a new node and update the pointer of the last element.
  - linked lists do not use excess memory like dynamic lists do.
- Cons:
  - access time is increased since linked-lists are not random access.
    - must traverse the entire list from the beginning to find what you want.
  - linked list elements are not guaranteed to be in contiguous memory locations.
  - linked lists can be challenging to manipulate
  - linked lists use some overhead memory for the pointers.

Linked Lists

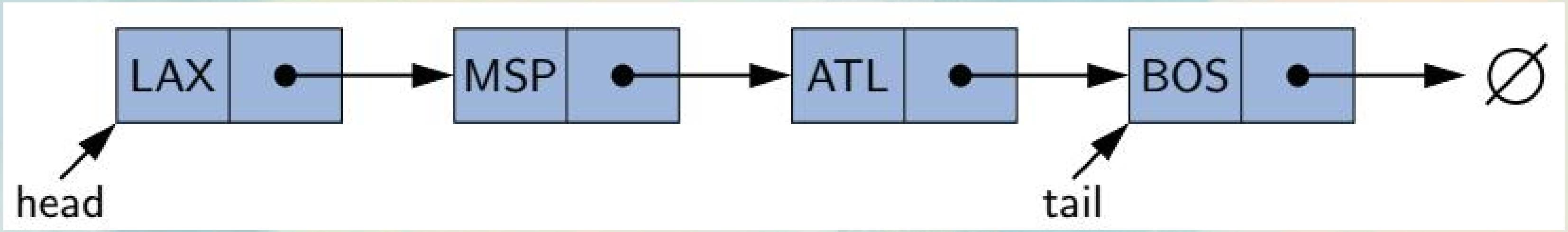


- Properties:
  - elements (nodes) are always connected by pointers (references)
  - the tail node usually points to NULL.
  - always has a *head pointer.*
  - sometimes has a *tail pointer.*
  - can increase or decrease its size dynamically
  - does not waste excess memory (but has a bit of overhead for each pointer.)

Singly-Linked Lists

Singly-Linked List

- The classic implementation of a Linked List is as a ***singly-linked list***
  - this is a linked list with only forward pointers.



  - definitely has a head pointer, but may or may not have a tail pointer.
  - last node always points to NULL.
  - may not do insertion / deletion from the middle of the list, however, these operations have been included for further study.
  - may not support indexing operations (manipulation using an index for finding an element)

Dynamic List ADT

- The ADT **SinglyLinkedList** is a linear sequence of an arbitrary number of items of the same type, which includes the following behaviors:
  - NOTE: This ADT will also have functions for isEmpty and size(). These will generally have the same implementation as the Dynamic List.

`SinglyLinkedList()`	Constructor which initializes an empty `SinglyLinkedList`.
`addFirst(item)`	Adds a new item to the beginning of the list.
`addLast(item)`	Adds a new item to the end of the list.
`delete(index)`	Deletes an item at the given index.
`deleteFirst()`	Deletes an the first item in the list.
`deleteLast()`	Deletes the last item in the list.
`get(index)`	Retrieves the item at the given index.
`getFirst()`	Returns the first item in the list.
`getLast()`	Returns the last item in the list.
`insert(index, item)`	Inserts an item at the given index.

DynamicList Class Overview

- NOTE: Again, a simple implementation to store only integers.

- `SinglyLinkedList` has two (maybe three) data fields:
  - `int size;` //specifying the current size of the list.
  - `Node head;` //pointer to the head of the list.
  - `Node tail;` //pointer to the tail of the list.
    - This pointer is optional, but its presence can help to make some functions more efficient.

The Node Class

```
class Node {
  private int data;
  private Node next;  //forward pointer

  public Node(int data) {
     this.data = data;
     this.next = null;
  }

  //Include any getters / setters as necessary.
}
```

Singly-Linked List Traversal

- Traversing a Linked List or *pointer hopping* means that you usually start from the head and follow the pointers to the item you are looking for.

- How do we know when we hit the end of the list?
  - The next pointer of the current node will have a value of **null**.

- The following code snippit shows a simple method which prints the data of each node.

```
current = head //start at at the beginning

while (current != null):
  print current.data
  current = current.next //hop to the next node
```

Singly-Linked List - get()

- `get(index)`:
  - Retrieves an item at the given index.
  - NOTE: The index is only to help us count hops, not to jump to the specified index.
- Runtime: O(n)
  - This method is always linear because you always have to traverse through the nodes on the average.

```
get(index):
  check if index is in bounds

  count = 0
  current = head

  while count < index:
    current = current.next
    count++

  return current.data
```

Singly-Linked List - addFirst()

- `addFirst(item)`:
  - Adds a new item to the beginning of the list.
  - Two cases to consider, the list is empty, or the list is not empty.

- Runtime: O(1)
  - This method is always constant because we always have a reference to the beginning.

```
addFirst(item):
  Node newNode = new Node(item);

  if isEmpty():
    head = newNode
    tail = newNode //if tail pointer exists
  else:
    newNode.next = head
    head = newNode

  size++
```

Singly-Linked List - addLast()

- `addLast(index)`:
  - Adds a new item to the end of the list.
  - Two cases to consider, the list is empty, or the list is not empty.
- Runtime: Varies
  - If you have a tail pointer: O(1)
  - If you do not have a tail pointer: O(n)

Singly-Linked List - addLast()

- `addLast(index)` with a tail pointer

```
addLast(item):
   Node newNode = new Node(item);

   if isEmpty():
      head = newNode
      tail = newNode
   else:
      tail.next = newNode
      tail = newNode

   size++
```

Singly-Linked List - addLast()

- `addLast(index)` without a tail pointer
  - Requires a full traversal of the list

```
addLast(item):
   Node newNode = new Node(item);

   if isEmpty():
      head = newNode
   else:
      current = head
      while current.next != null:
         current = current.next

      current.next = newNode

   size++
```

Singly-Linked List - insert()

- `insert(index, item)`:
  - Inserts an item at the specified index.
  - May not be included in most Linked Lists because of its runtime.
  - Requires a traversal of the list until we find the node directly before the insertion point.
  - Three cases to consider: inserting at beginning, end, or middle.
  - NOTE: The index is only to help us count hops, not to jump to the specified index

- Runtime:
  - Worst Case: O(n) (Node traversal)
  - Best Case: O(1) (Insertion to beginning or end (assuming tail pointer exists))
  - **Amortized Analysis:** O(n)

Singly-Linked List - insert()

```
insert(index, item):
  check for index out of bounds

  Node newNode = new Node(item)

  if index == 0:
    addFirst(item)
  else if index == size:
    addLast(item)
  else:
    current = head
    count = 0

    while count < index - 1:
      current = current.next
      count++

    newNode.next = current.next
    current.next = newNode
    size++
```

Singly-Linked List - deleteFirst()

- `deleteFirst()`:
  - Removes the first item in the list.
  - Things to consider: The list is empty, the list has one element, the list has more than one element.
- Runtime: O(1)
  - This method is always constant because we always have a reference to the beginning.

```
deleteFirst():

  if isEmpty():
    Error: list is empty, nothing to delete
  else:
    if size == 1:  //this part required
      tail = null  //  if tail pointer exists
    head = head.next
    size--
```

Singly-Linked List - deleteLast()

- `deleteLast()`:
  - Removes the item at the end of the list.
  - Things to consider, the list is empty, the list has one element, the list has more than one element, there is or is not a tail pointer.

- Runtime: Varies
  - If you have a tail pointer: O(1)
    - The list is doubly linked.
  - If you do not have a tail pointer: O(n)

Singly-Linked List - deleteLast()

```
deleteLast():

  if isEmpty():
    Error: Nothing to delete.
  else if size == 1: //required with tail pointer
    head = null
    tail = null
  else
    current = head
    previous = head
    while current.next != null:
      previous = current
      current = current.next

    previous.next = null
    tail = previous //required with tail pointer
  size--
```

Singly-Linked List - delete()

- `delete(index, item)`:
  - Deletes an item at the specified index.
  - May not be included in most Linked Lists because of its runtime.
  - Requires a traversal of the list until we find the node directly before the deletion point.
  - Three cases to consider: deleting from the beginning, end, or middle.
  - NOTE: The count is only to help us count hops, not to jump to the specified index
- Runtime: O(n)
  - Always requires a traversal because we need to find the next to last node in the list.

Singly-Linked List - delete()

```
delete(index):
  check for index out of bounds

  if index == 0:
    deleteFirst()
  else if index == size-1:
    deleteLast()
  else:
    previous = head
    count = 0

    while count < index:
      count++
      previous = previous.next

    previous.next = previous.next.next
    size++
```

Variations on Linked Lists

Tail or No Tail Pointer

- Tail or no Tail:
  - As shown previously, a linked list can optionally keep track of a tail pointer.
  - Keep in mind that as your list changes you ALWAYS have to watch for changes to the tail pointer and ALWAYS keep it up to date.

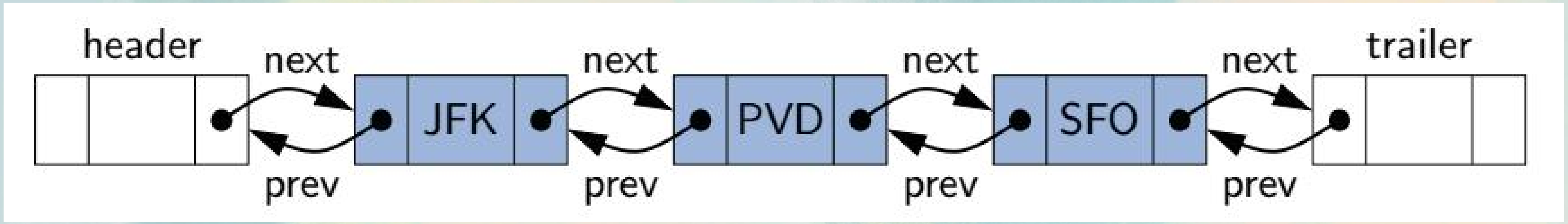
Singly or Doubly Linked List

- We have seen basic implementations of a ***singly linked list*** where each node only keeps track of a forward pointer.

- A variation of this is having a ***doubly-linked list*** where each node keeps a forward AND a backward pointer.
  - This requires you to be extra careful about updating the pointers correctly.
  - Allows you to traverse the list in reverse and can actually simply some of the previous implementations.
  - With a doubly-linked list, you always want a tail pointer.

Doubly-Linked List: Header and Trailer Sentinals

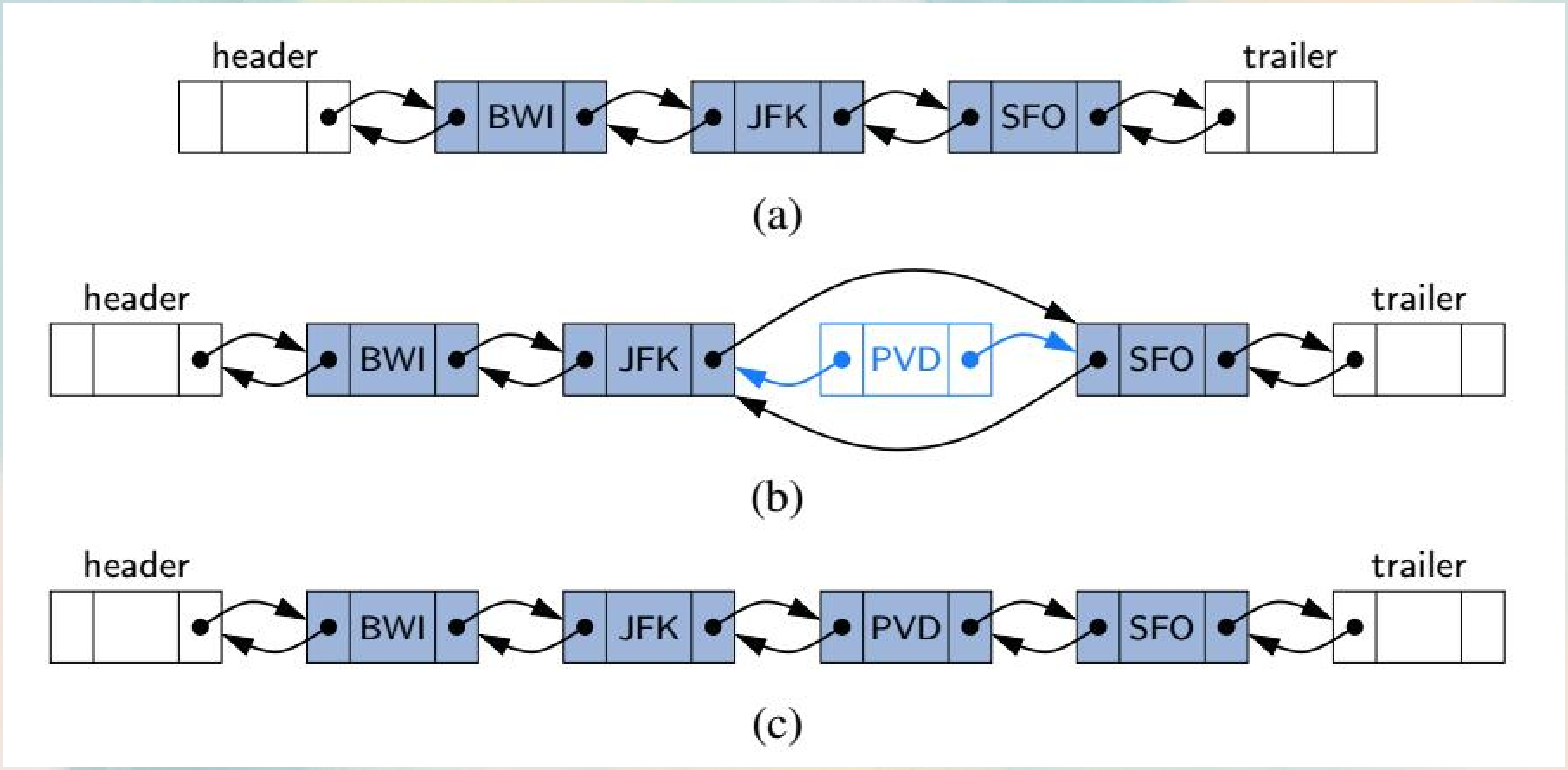
- To avoid special cases in a doubly-linked list when working near either end, it helps to have special nodes on both sides
  - a ***header*** node at the beginning of the list.
  - a ***trailer*** node at the end of the list



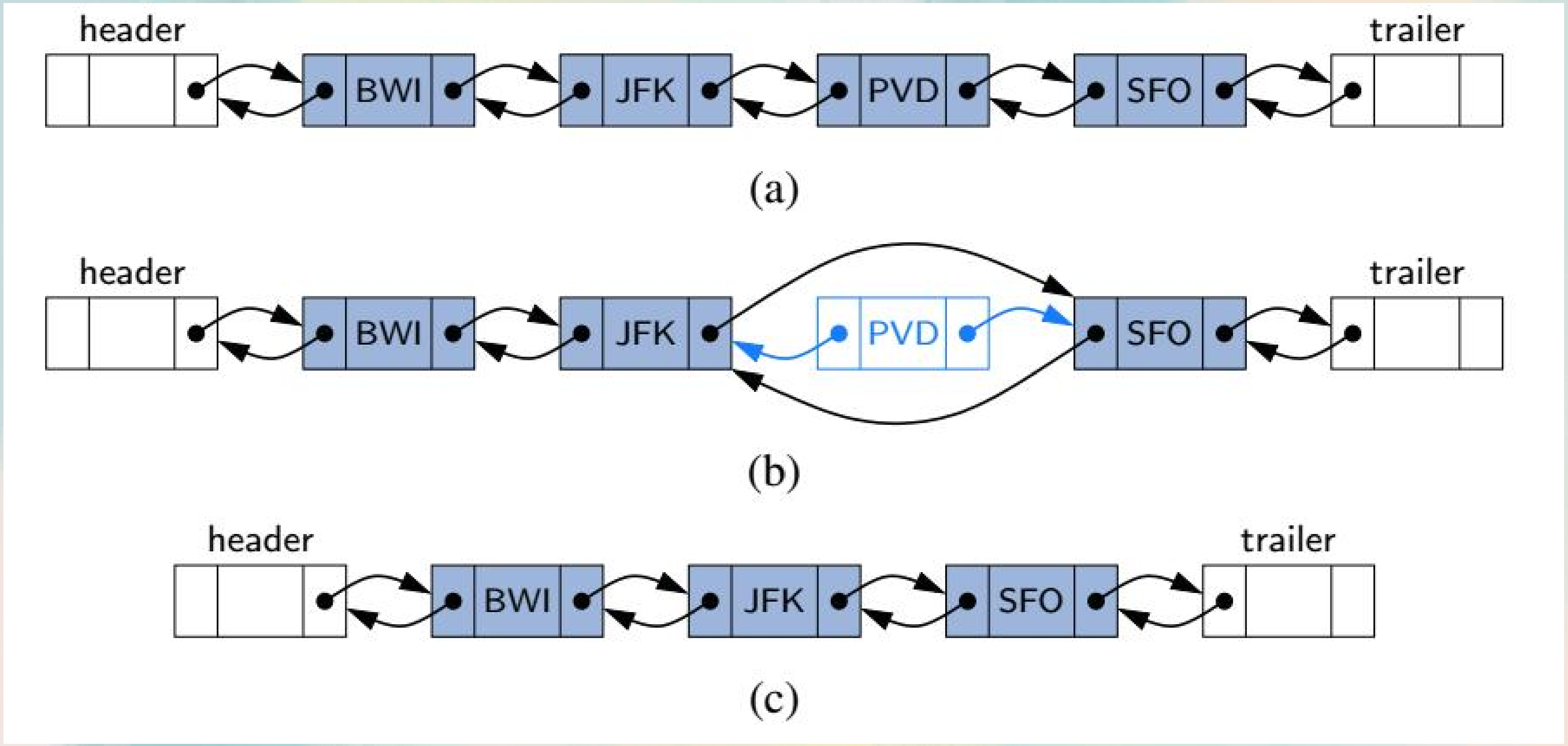
- These are dummy nodes that hold ***no values*** and are called sentinals or "guards" which help us to track when we are near the beginning or end.

- Advantages:
  - header and trailer nodes never change, only the nodes between them.
  - all insertions are treated in a unified manner (a new node is always placed between two other nodes.
  - all deletions are unified (a node to be deleted is guaranteed to have neighbors on either side).

Doubly-Linked List: Insertion



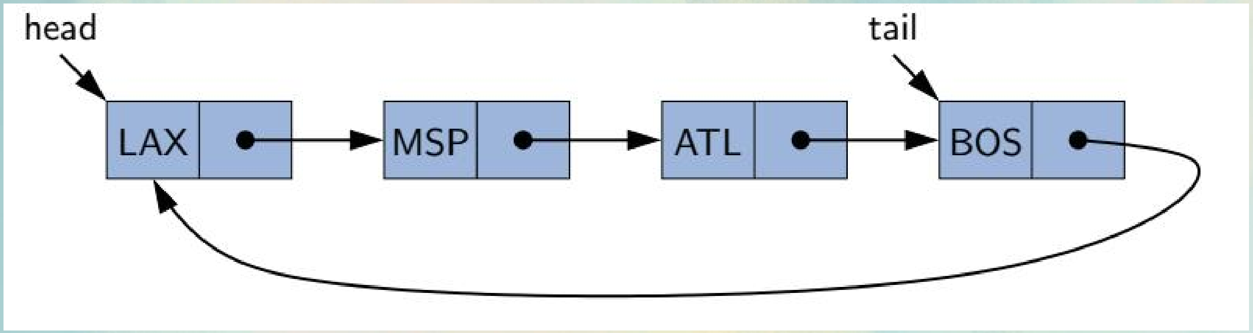
Doubly-Linked List: Deletion



Circular vs NonCircular

- So far we have been talking about ***non-circular linked lists***.

- Another variation is to have a ***circular linked list***
  - ***Circular Singly-Linked List:*** the forward pointer of the last node points back to the head.



  - ***Circular Doubly-Linked List:***
    - the forward pointer of the last node points back to the head.
    - the backwards pointer of the first node points back to the tail.

Choosing the Correct Type of List

- Use an array when:
  - The size of your data is fixed and does not change throughout the runtime of your program.
- Use a Dynamic List when:
  - The size of your data can change while the program is running.
  - You want random access of all elements in your List.
  - You do not need to worry about excess memory usage.
  - You want to sort your data.
- Use a Linked List when:
  - The size of your data can change while the program is running.
  - You need to worry about excess memory.
  - You are only only accessing items at the head / tail of the list.
  - You do not need random access.
  - You do not care about how the data is sorted.

References

- Goodrich, Chapters 03 and 07