Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 3

Dr Nick Hayward

Algorithms and Data Structures

Linked list - memory

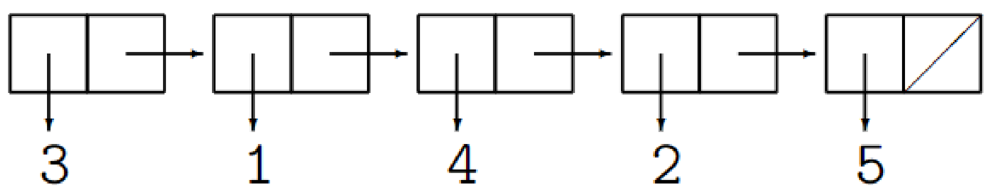
- a linked list
  - *data may be stored anywhere in memory*
  - *each item stores address of the next item in the list*
  - *i.e. link together random memory addresses as a contiguous structure*

.	.	.	.	.	.	.	.
.	X	.	.	.	X	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	X	.	.	.	.	.
.	.	.	.	X	.	.	.
.	.	.	.	.	.	.	.
.	X	.	.	.	.	.	.

- request an item from the list
  - *also returns the address of the next available item*
  - *like following a trail of clues to find the answers*
- with a linked list
  - *do not need to move items in memory*
- may also add a new item anywhere in memory
  - *then save address to previous item in the list*
- with a linked list, we do not need to move items

- i.e. if there’s space available in memory, there is space for a linked list *linked list - representation*

- also consider a *linked list* data structure to store our list of items in memory
- represent non-empty lists as *two-cells*
- each cell defined as follows
  - *first cell - contains pointer to a list element*
  - *second cell - contains pointer to empty list or another two-cell*
- structure is commonly represented graphically as follows



- pointer to empty list may be shown with a diagonal line
  - *crossing out the cell*
- list is a representation of `[3, 1, 4, 2, 5]`

linked list - inductive construct - part 1

- now consider a *linked list* using two *constructors*
  - `EmptyList` *- constructs the required empty list*
  - `MakeList(element, list)` *- puts element at top of existing, defined list*
- e.g. we may use these constructors as follows

```
MakeList(3, MakeList(1, MakeList(4, MakeList(2, MakeList(5, EmptyList)))))
```

- use to construct our previous list
- use this pattern of constructor execution to construct any list
- *inductive* approach to the creation of data structures is particularly useful
  - *easy to reason and follow.*
- e.g. start with *base case*, `EmptyList`, and
  - *then build more complex lists by repeating induction step,* `MakeList(element, list)`

Algorithms and Data Structures

linked list - inductive construct - part 2

- as with example array
  - *need a pattern to allow us to retrieve the list's elements*
  - *retrieve in a predictable and repeatable manner*
- unlike the array
  - *we do not have an item index*
- we may use known pattern of a list's construction
- i.e. a list is constructed
  - *from first element*
  - *the rest of the list*
- now know that for a non-empty list
  - *always get the first element and the rest...*
- now define two *accessor methods* for our lists
  - `first(list)`
  - `rest(list)`
- *accessors* or *selectors* only useful for non-empty lists
  - *throw an error for an empty list*
- add a condition to check if a given list is empty
  - `isEmpty(list)`
- then call it for each list before passing it to an *accessor* or *selector*

linked list - inductive construct - part 3

- now define a list constructed with `EmptyList` and `MakeList`
  - *including accessors `first` and `rest`*
  - *and the condition `isEmpty`*
- such that the following relationsips may be `true`
  - *`isEmpty(EmptyList)`*
  - *`not isEmpty(MakeList(x,l))` - holds for any `x` and `l` (l = list)*
  - *`first(MakeList(x,l)) = x`*
  - *`rest(MakeList(x,l)) = l`*
- also need to *destructively* change lists
- *Mutators* used to modify either first element or rest of a non-empty list
- e.g.
  - *`replaceFirst(x,l)`*
  - *`replaceRest(r,l)`*
- e.g. for `l = [3, 1, 4, 2, 5]` test the following
  - *`replaceFirst(7, l)` - modifies `l` to `[7, 1, 4, 2, 5]`*
  - *`replaceRest([3, 2, 6, 4], l)` - modifies `l` to `[7, 3, 2, 6, 4]`*
- predicated on expected patterns for first and rest with a list data structure
- concepts for *constructors*, *selectors*, and *conditions*
  - *common place for almost all data structures*
  - *help with abstraction of data types and algorithms*

Video - Algorithms and Data Structures

linked list - part 1

Data Structures: Linked Lists

Source - Linked Lists- YouTube

linked list - implementation

- as we use and design data structures for various algorithms
  - *may find differing implementations of same underlying conceptual outline*
- e.g. *lists*
  - *implementations may depend on primitives offered by a given programming language*
- *list* data structure in Python, Lisp, &c.
  - *considered important primitive data structure*
  - *filling same basic role as arrays in JavaScript...*
- often see same conceptual design either core to a language
  - *or based upon other data structures*
- issues with latter approach for some languages
- e.g. need to ensure that use of an array
  - *i.e. as a construct for a list*
  - *does not limit its size*
- role of defined selectors, mutators &c.
  - *ensure algorithm is implemented correctly with constructed list*

linked list - JS example - part 1

- consider an example implementation in JavaScript for a *linked list*
  - *& constructs required for an algorithm to ensure it functions as expected*
- begin by considering initial criteria for our custom *linked list*
  - *multiple values stored in a linear pattern*
  - *each value stored in a node*
  - *& a link to the next node in the list*
  - `null` *if no next node pointer required*
- initially consider a *singly linked list*
  - *i.e. a node has just one pointer to another node, or* `null`
- for JavaScript development
  - *might consider an array as a suitable data structure*
  - *perhaps for approximating linked list usage*
- whilst array size is dynamic
  - *still requires customisation to provide expected functionality of a linked list*
- example may use an array as the foundation
  - *and construct the linked list...*

linked list - JS example - part 2

- begin with initial design of node structure for a *linked list*
- each node in the list must contain some data and the pointer to the next node
- e.g. following code is a simple JS representation of this pattern

```
class LinkedNode {
  // instantiate with default props for linked list node
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
```

- constructor for this class
  - *defines default properties required for a Linked List node*
  - *data may be defined upon instantiation*
  - `next` *pointer is initially* `null` *- no pointer is yet available for this property*

linked list - JS example - part 3

- use this class as follows to create the first node in a list, which is customarily named `head`

```
// create first node in list
const head = new LinkedNode(13);
// create second node and assign to pointer
head.next = new LinkedNode(9);
```

Algorithms and Data Structures

linked list - JS example - part 4

- initial traversal follows an algorithm
  - *defined using the inherent structure of a linked list*
- algorithm allows an app to traverse all of the list's data
  - *simply by following `next` pointer defined for each node*

```
let currentNode = head;

while (currentNode !== null) {
  // get data for current node - output, send to DB &c...
  let data = current.data;
  // update pointer for current node
  current = current.next;
}
```

- traversal follows a simple pattern
  - *informed by structure of the linked list itself*
- traversal will continue until we reach end of current list
  - *and `current` is set to `null`*
- algorithm is a common example, regardless of langauge, for initial traversal of a linked list

linked list - JS example - part 5

- also define a class to work with overall Linked List
  - *not just individual nodes*

```
class LinkedList {
  constructor() {
    ...
  }
}
```

- to ensure `head` node is always unique to this Linked List
  - *use a recent JS data type* `Symbol`
- `Symbol` added to JavaScript with ES2015 (ES6)
  - *other benefit is useful description for variable as part of declaration*

```
const head = Symbol('head');

class LinkedList {
  constructor() {
    // set initial pointer to first node in list
    this[head] = null;
  }
}
```

- description is useful for debugging and monitoring variable
  - *and its usage*
- may not be used to access the Symbol itself
- we’ve now set initial pointer for linked list

linked list - JS example - part 6

- after creating initial empty Linked List
  - *need to define a method to allow us to add a new node*
- adding some new data to a linked list
  - *requires traversing structure to find a suitable location*
  - *create the node*
  - *insert in identified location in list*
- if list is empty
  - *simply create new node and assign it to* `head` *of list*

```
addNode(data) {
  // create new node
  const newNode = new LinkedNode(33);
  // handle empty list - no items
  if (this[head] === null) {
    // head set to new node
    this[head] = newNode;
  } else {
    // look at first node
    let current = this[head];
    // follow pointers to the end...
    while (current.next !== null) {
      // update current
      current = current.next;
    }
    // update node for next pointer
    current.next = newNode;
  }
}
```

linked list - JS example - part 7

- in previous example code
  - *`addNode()` method defines a single parameter - data for node*
  - *then adds it to end of list*
- we check list - if it is empty
  - *i.e. `head is null`*
  - *assign new node as `head` of list*
- if list has existing nodes
  - *need to traverse it to reach end - the last node*
- uses a simple `while` loop
  - *starting at `head`*
  - *following `next` pointers until we find last node*
- last node will have its `next` pointer set to `null`
  - *stop traversal at this point*
  - *ensure we don't update `current` to `null`*
- allows us to assign new node to `next` pointer
  - *adding data to current list*

Video - Algorithms and Data Structures

linked list - part 2

Data Structures: Linked Lists

Source - Linked Lists- YouTube

issues with linked lists

- a linked list may seem a preferable solution
  - *we may still encounter noticeable issues with such lists*
- e.g. if we need to access item 10 in a linked list
  - *need to know the address location in memory.*
  - *need to get the address from the previous item in the linked list*
- that item needs to get the address from the previous item
  - *and so on to the first item in the linked list…*
- quickly see that a linked list is great
  - *if we need to access each item one at a time*
  - *may read one item, then move to the next item, and so on…*
- if we need to access items out of order
  - *on a regular basis*
  - *a linked list is a poor choice*

benefits of arrays

- accessing an array is a noticeable benefit compared to a linked list
- address known for every item in the array
  - *quickly and easily access an indexed item*
- arrays are a good option if we need to access random items on a regular basis
  - *easily calculate the position of an array item*
  - *contrasts strongly with rigid pattern of access for a linked list*

runtime comparison - part 1

- comparative run times for common operations on arrays and lists

	Arrays	Lists
reading	O(1)	O(n)
insertion	O(n)	O(1)

- key:
  - *O(n) = linear time*
  - *O(1) = constant time*
- *linear time* for *array* insertion and *list* reading
- constant time for *array* reading and *list* insertion

insertion in the middle

- may need to modify our data storage for an app
  - *e.g. to allow insertion in the middle of the data structure*
- our choice of array or linked list will also affect this option
  - *and the efficiency of insertion*
- e.g. if we consider a linked list
  - *it's as easy as modifying address reference of previous element to point to inserted data item*
- for arrays
  - *need to shift all of the remaining elements down to create space for the inserted items*
  - *if there is not sufficient space in the current address location*
    - may also need to move whole array before we can insert new items
- may see a performance benefit for insertion to middle of a linked list compared to an array

deletions

- which option is preferable for deletion?
- for most use cases
    - *simpler to delete an item from a linked list*
    - *only need to modify address reference for previous item in the list*
- for an array
    - *again, need to move the whole array to accommodate the deletion*

runtime comparison - part 2

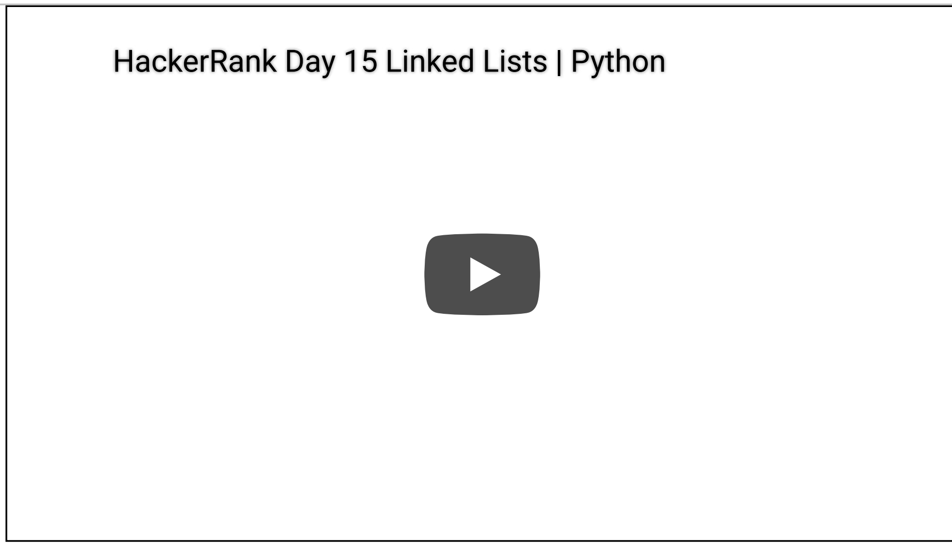
- update our comparison table to now include *delete* operations for both arrays and linked lists

	Arrays	Lists
reading	O(1)	O(n)
insertion	O(n)	O(1)
deletion	O(n)	O(1)

- key:
  - *O(n) = linear time*
  - *O(1) = constant time*
- it's worth noting
  - *both insertions and deletions may be* `O(1)` *run time - only if we may access the element instantly*
- e.g. common practice in such algorithms to maintain a record of the first and last items in a linked list
  - *then only take* `O(1)` *run time to delete such items*

Video - Algorithms and Data Structures

sample linked list question



HackerRank Day 15 - Linked Lists - Python

Source - Linked Lists - YouTube

Algorithms and Data Structures

linked list - JS example - part 8

- update our JavaScript example
  - *include getting and deleting data from linked list*
- e.g. consider getting data from list
  - *update code with `getNode()` method for `LinkedList` class*
- method allows us to get data for a node
  - *in any given position in the list using traversal*

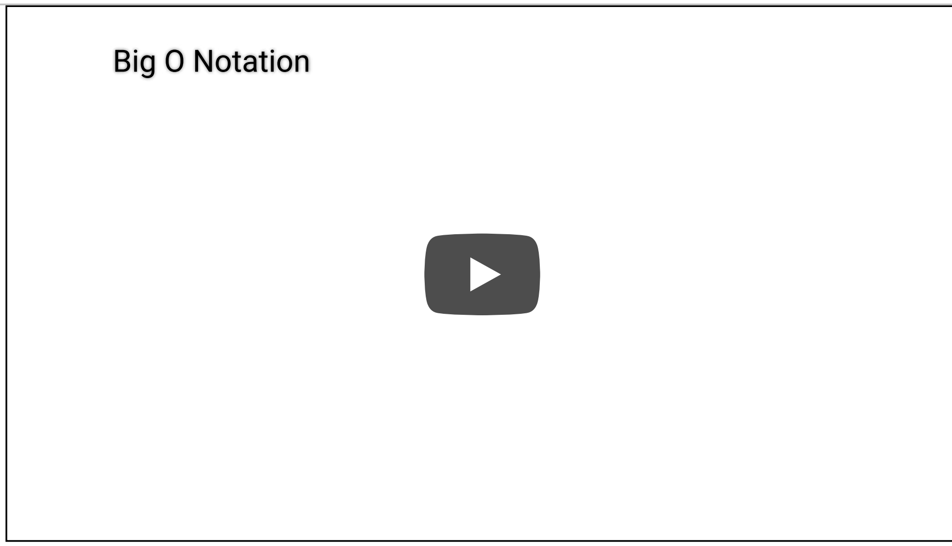
```javascript
// traverse list to defined index posn
getNode(index) {
  // check index value is positive
  if (index > -1) {
    // initial pointer for traversal
    let current = this[head];
    // record location in list...
    let i = 0;
    // traverse list - until either index or end
    while ((current !== null) && (i < index>)) {
      // update current
      current = current.next;
      // increment location
      i++;
    }
    // return data - i.e. current != null
    return current !== null ? current.data : undefined;
  }
  else {
    return undefined;
  }
}
```

linked list - JS example - part 9

- `getNode()` method initially checks requested `index` value is positive
- if not
  - *simply return `undefined`*
  - *perhaps, try again...*
- `index` is valid
  - *traverse list*
  - *maintain record of current location in list*
- `while` loop has similar logic to earlier method for addition
  - *we may now also exit when current location equals required `index`*
- complexity of `getNode()` method may range from
  - *`O(1)` when removing first node (i.e. no traversal necessary...) to*
  - *`O(n)` when removing the last node (i.e. traversal of the complete list)*
- based on simple fact that we always need to perform a search to find correct value

Video - Big O notation

fun refresh - part 1

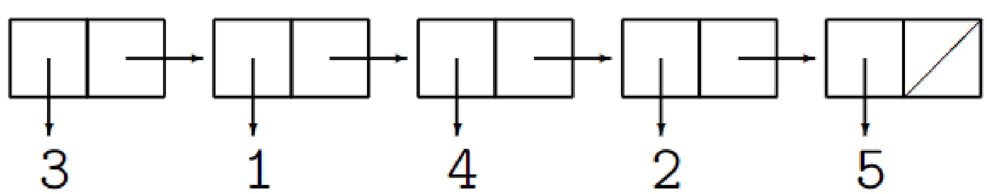


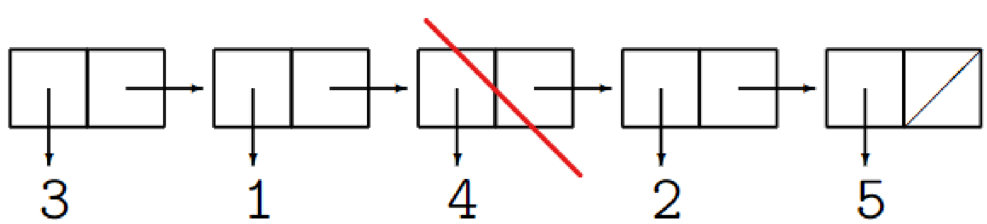
A fun reminder of Big O Notation

Source - Big O Notation - YouTube

linked list - JS example - part 10

- also need to consider how we may delete data from our linked list
- deleting data from a linked list may be a tad involved...
- need to check and ensure all `next` pointers remain valid
  - *i.e. after deletion has been executed*
- e.g. we need to ensure that `next` pointer is updated
  - *and identifies correct node in the list*
- if we have a list of 5 nodes
  - *then delete node 3*
  - *node 2 needs to point to previous node 4*





- i.e. we have to consider two operations for the deletion
  - *find position of specified node in list*
    - same algorithm as `getNode()`

- *delete node at that position*

linked list - JS example - part 11

- underlying algorithm for finding the node
  - *i.e. to delete in the specified linked list*
  - *same as the `getNode()` method*
- we also need to check and record pointer for previous node
  - *i.e. it will need to be updated*
- e.g. delete node 3
  - *keep a record of node 2*
  - *modify pointer for node 2 to node 4...*
- we also need to consider following special cases for this delete operation
  - *list is empty - no traversal*
  - *index is less than 0 - i.e. invalid index...*
  - *index is greater than number of items in list*
  - *index is zero - removes `head` from list*

linked list - JS example - part 12

An example implementation for the method `deleteNode()`,

```
// delete node at specified index posn in list
deleteNode(index) {
  // check against special case - empty list, invalid index
  if ((this[head] === null ) || (index < 0)) {
    // throw error - index not in list...
    // e.g log error, return message, throw range error &c.
  }
  // check against special case - removing first node
  if (index === 0) {
    // store data from node
    const data = this[head].data;
    // update head with next node in list...
    this[head] = this[head].next;
    // return data stored before update
    return data;
  }
  // define pointer for list traversal...
  let current = this[head];
  // track previous node before current...
  let previous = null;
  // track depth of list...
  let i = 0;

  // traverse list - until either index or end
  // same basic loop as `getNode()`
  while ((current !== null) && (i < index)) {
    // store value of current
    previous = current;
    // update current
    current = current.next;
    // increment location
```

```
    i++;
  }

  // if node found - delete
  if (current !== null) {
    // modify pointer to skip current - delete from list
    previous.next = current.next;
    // return deleted node's value
    return current.data;
  }
  // throw error - node not found...
  // e.g log error, return message, throw range error &c.
}
```

n.b. explanation on next slide...

Algorithms and Data Structures

linked list - JS example - part 13

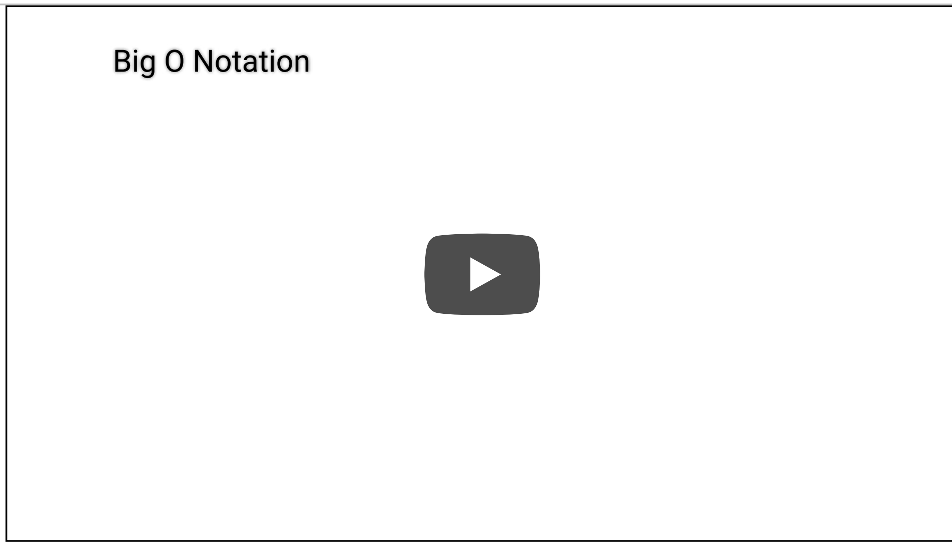
- `deleteNode()` method initially checks for two defined special cases
  - *empty list - `this[head] === null`*
  - *index less than zero - `index < 0`*
- for each of these cases
  - *we may throw an error*
  - *handle error appropriate to current app*
- then check for a case when `index === 0`
  - *i.e. check removal of `head` of list*
- new `head` will now become current second node in list
  - *requires a simple update of `head`*
  - *update to its current next pointer*
  - *i.e. `this[head].next`*

linked list - JS example - part 14

- also handle removal of a single node list
  - *returns* `null`
- list will become empty after the executed deletion
  - *need to ensure node's data is saved*
  - *i.e. use node's data after deletion*
- then traverse list with same basic pattern of iteration
  - *same pattern as* `getNode()` *method*
- main difference is record of `previous`
  - *tracks node before* `current`
  - *necessary for correct deletion of node*
- same manner as `getNode()`
  - *loop may exit with* `current` *as* `null`
  - *indicating index was not found*
  - *may throw error and handle...*
- then return any data stored in `current`

Video - Big O notation

fun refresh - part 2



A fun reminder of Big O Notation - UPTO 4.36

Source - Big O Notation - YouTube

linked list - JS example - complexity

- complexity for `deleteNode()` is same as `getNode()`
- range from **O(1)** (constant time) to **O(n)** (linear time)
- **O(1)** (constant time)
  - *as we remove the first node*
- **O(n)** (linear time)
  - *for removal of the last node*
- we may conceptually improve performance of these methods
  - *perhaps modifying the way we work with the list*
- e.g. both insertions and deletions may be `O(1)` run time
  - *only if we may access the element instantly*
  - *e.g. the first node*
- in such algorithms - maintain a record of first and last items
  - *then only take `O(1)` run time to delete such items*
- e.g. keep a record for such usage in the `head` and `tail`

Algorithms and Data Structures

linked list - JS example - part 15

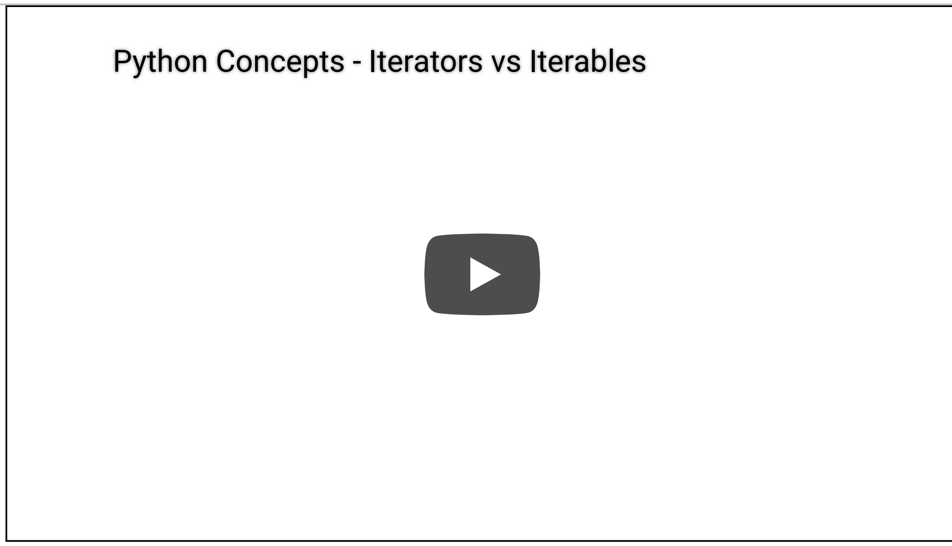
- our custom linked list is not currently iterable by default
- in JavaScript - a plain object is not iterable by default
  - *add a custom iterator for the object*
- might use a built-in custom object
  - *such as an Array*
  - *includes an iterator by default*
  - *one of the differentiating factors between a JS Array and a plain object*
- for this custom data structure - linked list
  - *need to make the object iterable*
- specify a custom iterator using JavaScript's built-in `Symbol.iterator`

linked list - nature of iterable

- nature of *iterability* may be defined as follows
  - *data consumers*
  - *data sources*
- JS has various language constructs to consume data, e.g.
  - *`for-of` loops over values*
  - *spread `(...)` operator inserts values into Arrays or function calls*
  - *…*
- JS may consume values from a variety of data sources, e.g.
  - *iterating element of an array*
  - *key/value entries in a Map*
  - *or simply the characters in a String*
  - *…*
- ES2015 (ES6) introduces an interface pattern for `Iterable`
  - *data consumers use it, data sources implement it…*

Video - Iterators vs Iterables

Python - part 1



Python - Iterables - UPTO 2.32

Source - Iterators vs Iterables - YouTube

Algorithms and Data Structures

linked list - traversing data - part 1

- relative to JS iteration
  - *commonly consider two parts to traversing data*
- *iterable*
  - *data type, structure to provide iterable access to the public*
  - *achieved with a method* `Symbol.iterator`
  - *a factory for iterators*
- *iterator*
  - *a pointer for traversing elements in a data structure*
- for a custom function
  - *we may return* `an iterable object with an iterator`
  - *return an iterator for an iterable...*

linked list - traversing data - part 2

We have standard built-in options for traversal, including

- destructuring via an array pattern

```
// destructuring via an Array pattern
const [a,b] = new Set(['hello', 'world', '!']);
// returns array of values from Set...
console.log([a,b]); // outputs ['hello', 'world']
```

- for-of loop

```
// for-of loop usage
for (const x of ['hello', 'world']) {
  // returns each array value...
  console.log(x);
}
```

- Array.from()

```
// Array.from
const arr = Array.from(new Set(['hello', 'world']))
// returns standard array - iterable as usual
console.log(arr[1]);
```

- Spread operator (. . .)

```
// Spread operator
const arrSpread = [...new Set(['hello', 'world'])];
// takes dynamic no. of values and returns array...
console.log(arrSpread);
```

linked list - traversing data - part 3

and some more options,

- constructors of Map and Set

```
// Map constructor - standard key/value pairings...
const testMap = new Map([[false, '0'], [true, '1']]);
// maps false to 0 &c.
console.log(testMap);
// use standard Map methods - e.g. get and set
console.log(testMap.get(false));
// use iterator methods
console.log(testMap.keys());
console.log(testMap.values());
// then iterate over returns from iterator method
console.log(Array.from(testMap.keys())); // returns expected array containing map
        values...
```

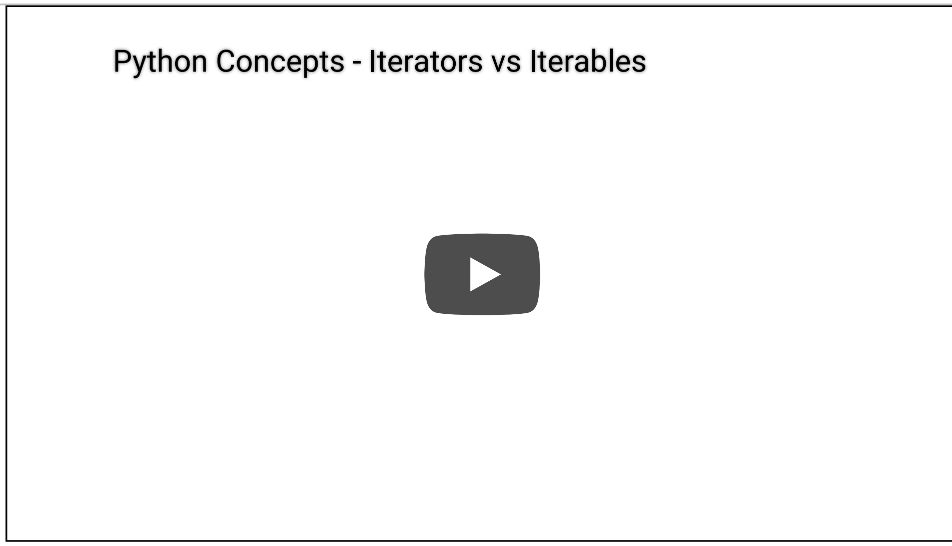
- and the same with `Set`

```
// Set constructor
const testSet = new Set(['hello', 'world']);
```

- and default iterables provided by Promise methods, e.g.
  - `Promise.all`
  - `Promise.race`
  - `yield*` *for generator iterables*

Video - Iterators vs Iterables

Python - part 2



Python - Iterators - UPTO END...

Source - Iterators vs Iterables - YouTube

linked list - implementing iterables

- many different ways we may work with existing iterable constructs
- e.g. manually define a custom iterator for an iterable object
- use a custom iterator with the Linked List
- need to add a custom *generator* method
- allows us to define how the object will be iterated
  - *the effective traversal of the linked list...*

linked list - JS example - part 16

- define and implement our custom iterator as follows

```
/*
* custom iterable
*   - generator method with custom iterator
*   - default iterator for class
*/
*[Symbol.iterator]() {
  // define start of iterator
  let current = this[head];
  // whilst nodes in linked list - until tail
  while (current !== null) {
    // yield each node's data
    yield current.data;
    // update current to next node
    current = current.next;
  }
}
```

linked list - JS example - part 17

- custom linked list data structure
  - *now iterable using standard built-in options for traversal*
- e.g. log to `console` using *spread* operator

```
console.log(...list);
```

- or with a `for...of` loop

```
// log all nodes in current list
for (const node of list) {
  console.log(node);
}
```

linked list - JS example - part 18

In a sample app, we may then use this linked list as follows,

```
// instantiate a new linked list
const list = new LinkedList();

// add some initial nodes to the linked list
list.addNode('castalia');
list.addNode('waldzell');
list.addNode('mariafels');

// get a specified node, and log to the console...
console.log('get node = ', list.getNode(1));

// log all nodes in current list
for (const node of list) {
    console.log(node);
}

// delete specified node from list
console.log('delete node = ', list.deleteNode(1));

// check linked list - spread nodes
console.log('spread updated list = ', ...list);
```

linked list - JS example - part 19

- our custom linked list data structure
  - *a class to instantiate a linked list -* `LinkedList`
  - *a class to instantiate a node in the linked list -* `LinkedNode`
  - *a method to add a node -* `addNode()`
  - *a method to get a node -* `getNode()`
  - *a method to delete a node -* `deleteNode()`
  - *defined a custom iterator for iterable Linked List object*
- may consider adding other methods, e.g.
  - *a counter for the total number of nodes*
  - *insert a node relative to a specific existing node*
    - before or after
  - *clear the linked list - i.e. delete all nodes*
  - *return index of defined node*
  - *...*

Video - NumPy

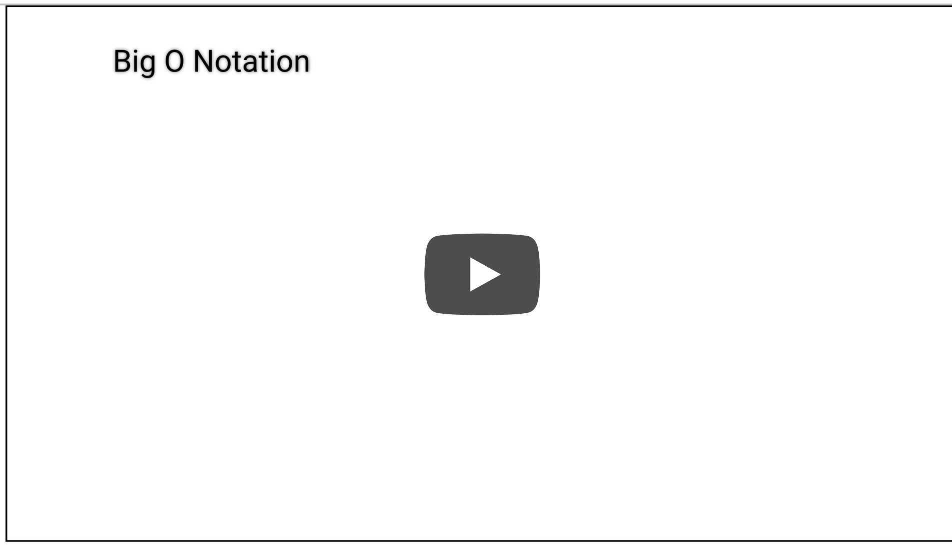
practical consideration of array vs list

A practical consideration of arrays vs lists in Python’s NumPy

Source - Lists vs Arrays, NumPy - YouTube

Video - Big O notation

fun refresh - part 3



A fun reminder of Big O Notation - UPTO END OF VIDEO

Source - Big O Notation - YouTube

Algorithms and Data Structures

Big O - simplify

- calculating time complexity is rarely as simple as counting number of loops in an algorithm
- e.g. if algorithm is
  - $O(n + n^2)$
- consider the following to help simplify complexity consideration

drop the constants

- e.g. an algorithm described as
  - $O(2n)$
- we may drop 2
- now becomes
  - $O(n)$

drop non-dominant terms

- we might have an example algorithm
- initially algorithm is as follows
  - $O(n^2 + n)$
- we may now drop n leaving
  - $O(n^2)$
- i.e. we keep the larger n^2 in Big O

caveat

- there are exceptions to such a rule
- e.g. if we have a sum

- *$O(b^2 + a)$*

- we may **not** simply drop either a or b
  - *without knowledge of them in this context*

n.b. for Big O notation, we often consider the following

> *What is the worst-case scenario?*

Algorithms and Data Structures

general usage preference - array vs linked list

- after considering both data structures
  - *which option is more commonly used for app development?*
- context is, of course, a valid consideration when choosing a data structure
- e.g. *arrays* may see frequent use due to their support for easy random access of data items
- these data structures support two initial types of access, *random* and *sequential*
- *sequential* access provides each data item in a consistent, predictable order
  - *exactly what we see when accessing a linked list data structure*
  - *i.e. only way to conveniently access data in a linked list*
- another benefit of random access
  - *a speed improvement in reading data*
  - *helps improve the performance of array data structures*
- may also see both arrays and linked lists used as the foundations for other, often specialised data structures…

Resources

JavaScript

- MDN - Classes
- MDN - Loops and Iteration
- MDN - Prototype
- MDN - Proxy
- MDN - Symbol
- MDN - Symbol.iterator

Various

- Big O Notation - YouTube
- Iterators vs Iterables - Python - YouTube
- Linked Lists - Java - YouTube
- Linked Lists - Python- YouTube
- Lists vs Arrays, NumPy - YouTube