Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 3

Dr Nick Hayward

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

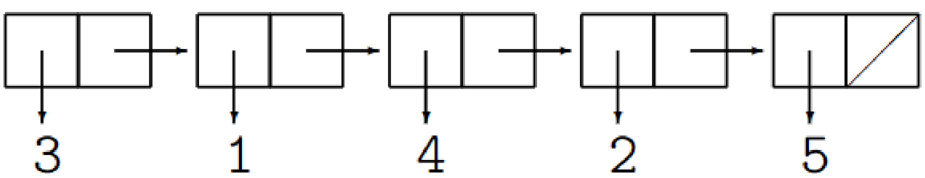
A fun reminder of Big O Notation

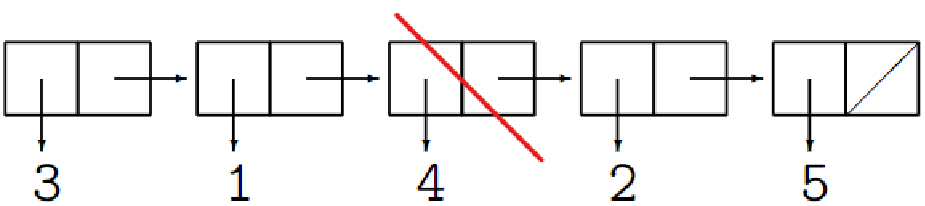
Source - Big O Notation - YouTube

Algorithms and Data Structures

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...

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`

Algorithms and Data Structures

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`

linked list - JS example - part 15

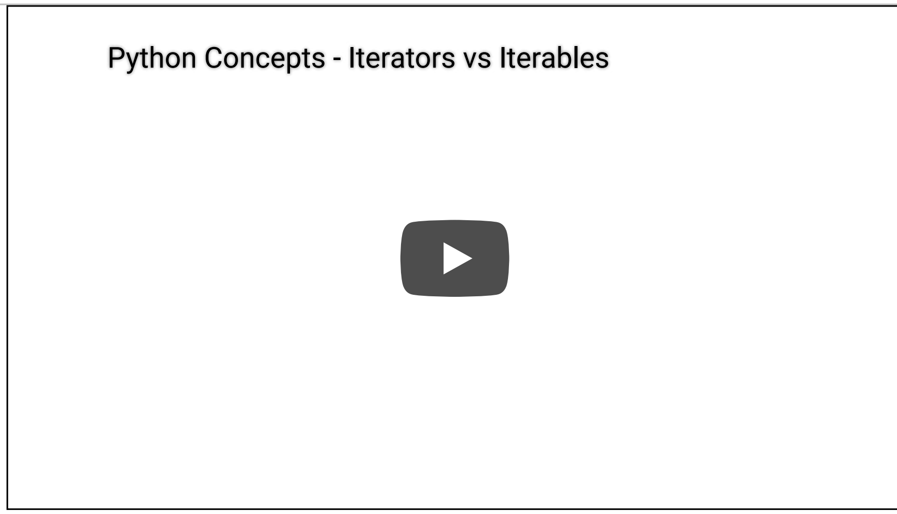
- our custom linked list is not currently iterable by default
- in JavaScript - a plain object is not iterable by default
  - *add a custom iterator for the object*
- might use a built-in custom object
  - *such as an Array*
  - *includes an iterator by default*
  - *one of the differentiating factors 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

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...
```

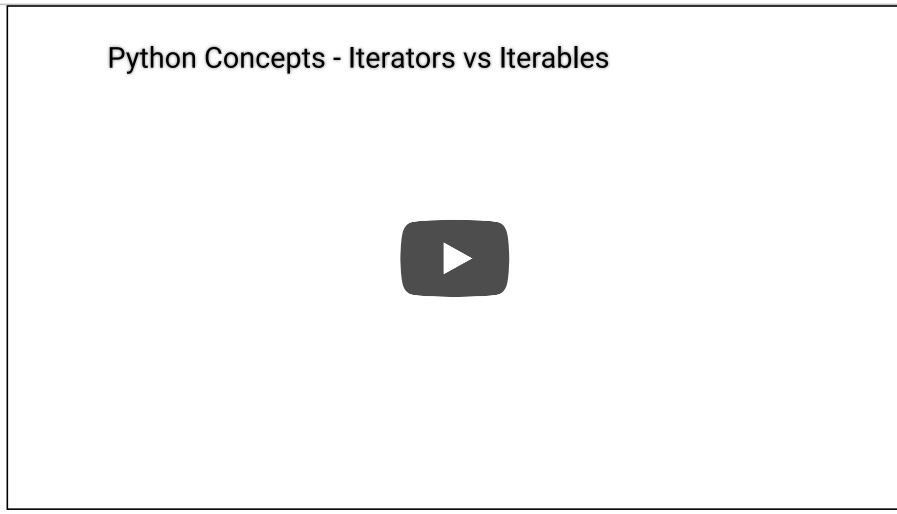
- and the same with `Set`

```
// 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 - 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

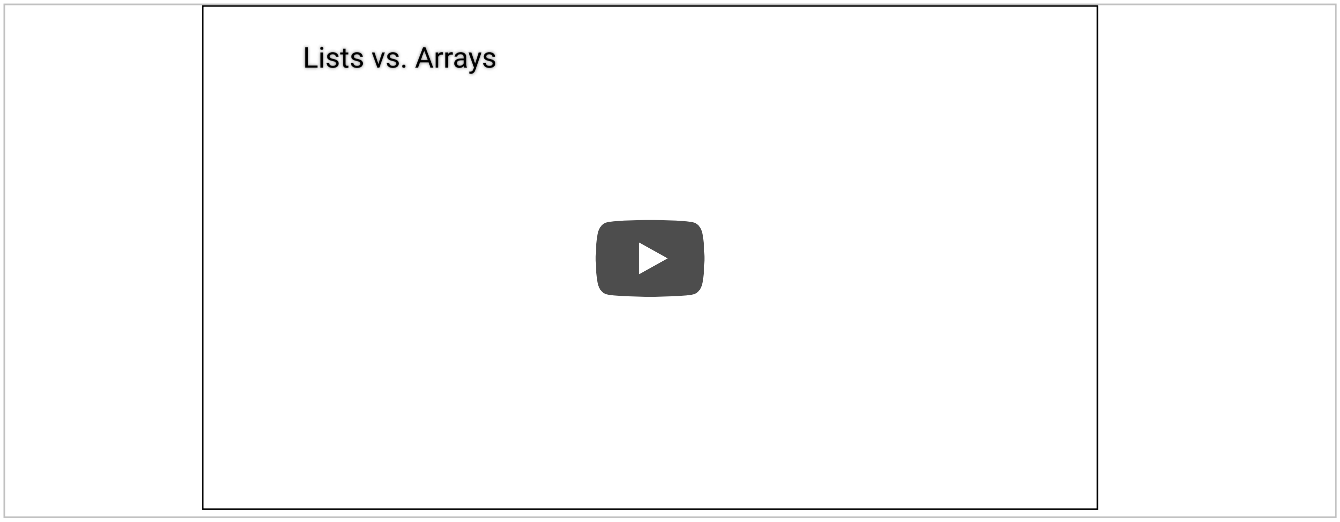
> *What is the worst-case scenario?*

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…

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 *pseudocode game*

Consider the following Snake game,

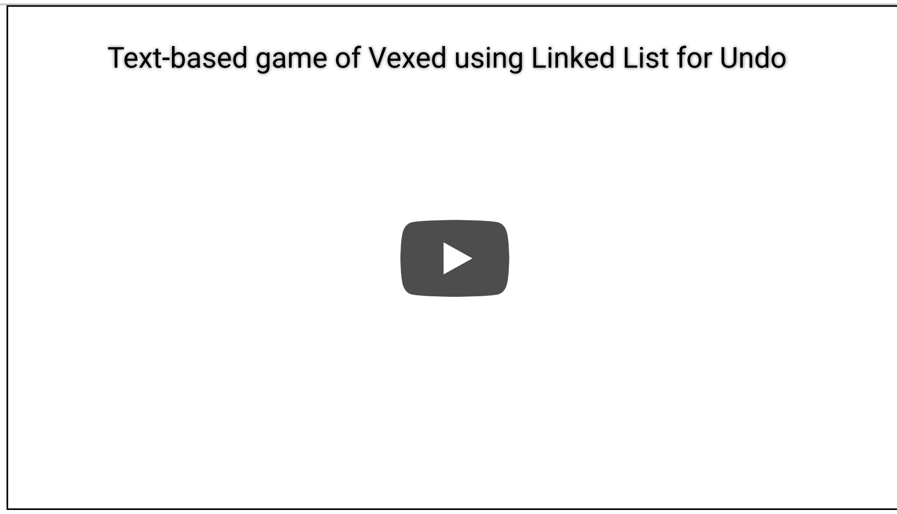
Then, using **pseudocode**

- define logic for this game
  - *use linked list*
- how will the following be used in this game
  - *accessors/selectors*
  - *mutators*

Approx. 10 minutes...

Video - Algorithms and Data Structures

Linked list in games



Text based game of Vexed

Source - Text based game with linked list - YouTube

System and Memory

app to memory

- as we design an appropriate algorithm,
  - *need to consider how an OS and application handle and use memory*
- e.g. an OS's management of memory
  - *closely associated with process requirements and usage*
- memory management relative to a process (e.g. application) may be considered broadly as follows
  - *ensure each process has enough memory to execute*
    - cannot run out of memory or use other processes' memory allocation
  - *different memory types must be organised efficiently*
    - ensures effective management of each process
- we may start by managing memory boundaries for different processes

System and Memory

process and memory usage

- if we consider restrictions and limitations of array implementation and management
  - *need a way to effectively manage this use of memory*
- as a child process is created, it is assigned an address memory space
- each process will see their memory space as a contiguous logical unit
- such memory addresses might actually be separate across the system
- disparate addressed memory spaces for each process
  - *may then be organised together, as needed, by the system's kernel*
- e.g. separate memory stores and addresses organised into a contiguous group per process
- benefit is efficient use of memory space
- no need for pre-assigned large chunks of memory
  - *or reserved memory that is never used by a process*
- kernel controls access for a process to memory addresses
  - *kernel is controlling conversion of assigned virtual addresses*
  - *converts to a physical address in the system's hardware memory*

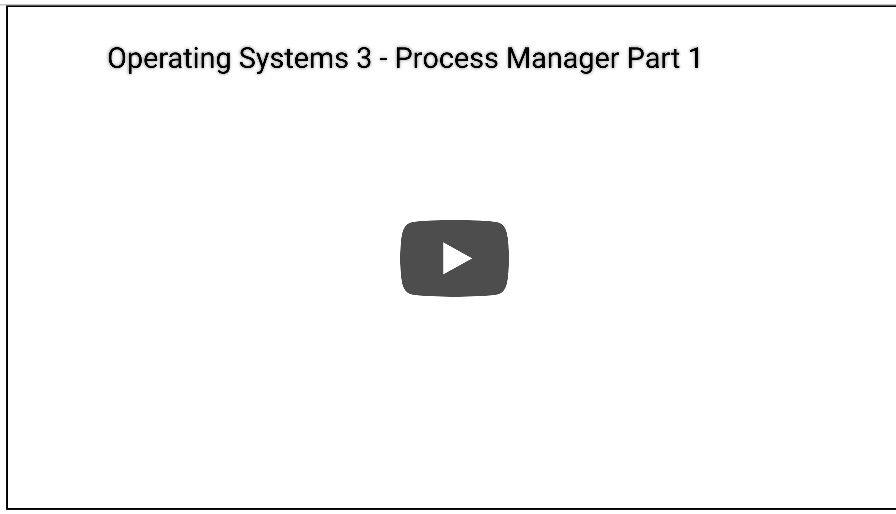
System and Memory

process and state

- each child process may have a related *state*
  - *associated during the lifetime of the process itself*
- state may be monitored by the system's *kernel*
  - *a process will wait until resources are available to allow a change in state*
- kernel may then switch processes relative to an update in a process' state

Video - System and Memory

process manager - part 1



Operating Systems - Process Manager - UPTO 1.41

Source - Process Manager - YouTube

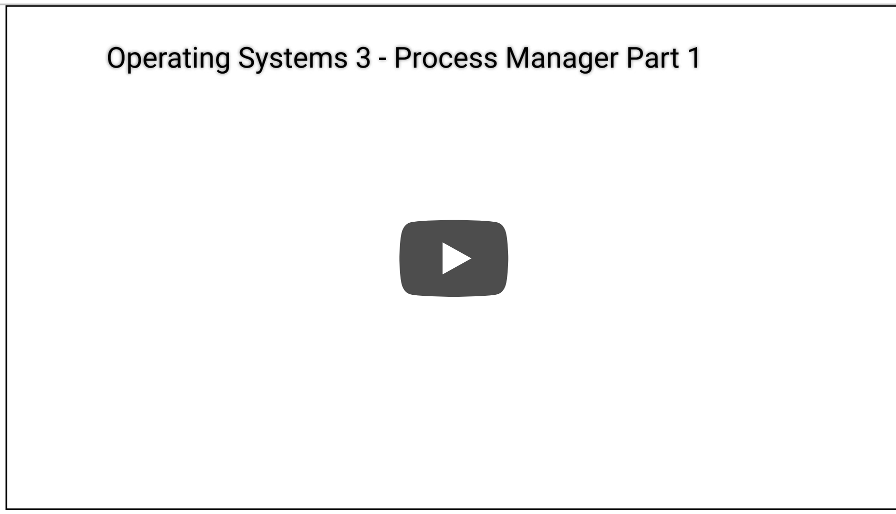
System and Memory

process manager

- *process manager* is responsible for processes in a system
- it is controlled by the system's *kernel*, and manages the following
  - *process creation and termination*
  - *resource allocation and protection*
  - *cooperation with device manager to implement I/O*
  - *implementation of address space*
  - *process scheduling*

Video - System and Memory

process manager - part 2



Operating Systems - Process Manager - Scheduler - UPTO END

Source - Process Manager - Scheduler - YouTube

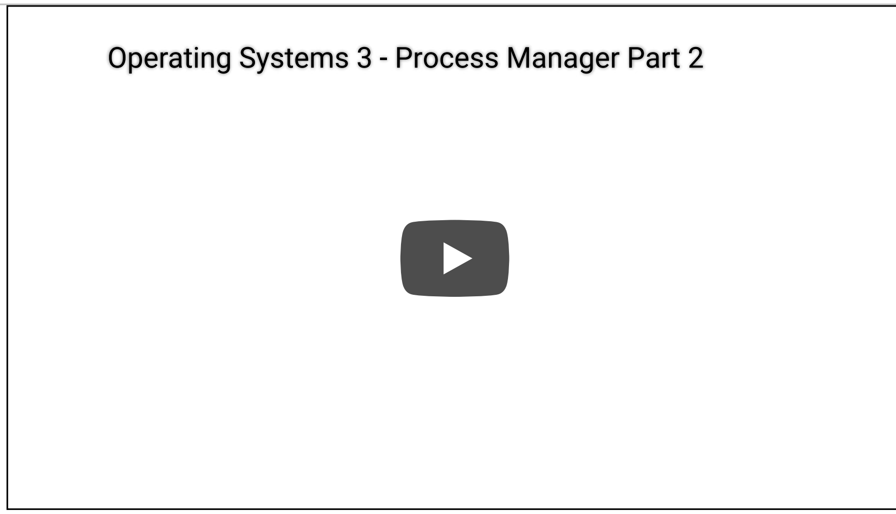
System and Memory

process scheduling

- a key part of managing processes in a system is efficient scheduling
- scheduling is part of the *process manager*
  - *actually maintained by the system's kernel*
- kernel is responsible for switching between processes
  - *checking and migrating available ready state processes to execution in an active state*
- *kernel* is selecting processes to execute in the system on the available CPU
  - *kernel is choosing the next process to run on the CPU*
- context switch is informed by the required and available *process properties*
- selection of process is also determined by the nature of the process itself
  - *i.e. is it I/O bound or CPU bound*
- algorithm helps determine the best process choice
  - *ensures system runs efficiently and without apparent delays*
- example algorithms include,
  - *first-come, first-served*
  - *shortest job next*
  - *round robin*
  - *multi-level priority queue*
- scheduling is meant to provide a fast and efficient system
- kernel chooses processes to allow the system to run fast
- e.g. it is common to assign priority to a *front-facing* process over one running in the *background*

Video - System and Memory

process manager - part 3



Operating Systems - Process Manager - Algorithms and Management

Source - Process Manager - Algorithms and Management - YouTube

Practical usage - JavaScript Arrays

intro

- collections in JS includes arrays
  - *associated built-in array methods, plus ES6 updates for sets and maps &c.*
- arrays in JS are simply objects
- as objects, arrays can access methods...

Practical usage - JavaScript Arrays

create an array

- two fundamental ways to create new arrays:
  - *using the built-in Array constructor*
  - *using array literals [ ]*
- e.g.

```
const readers = ["emma", "yvaine", "daisy"];
const archives = new Array("waldzell", "mariafels");
```

- array literals tend to be the more common option for JS development
- **n.b.** Writing to indexes outside the array bounds extends the array
- e.g. `readers.length === 5`
- if we try to write to a position outside of array bounds, as in

```
readers[4] = "bea";
```

- array will expand to accommodate the new situation
- may end up creating a hole in the array
  - *the item at index 3 will be `undefined`*
  - *length property will also be updated*

Practical usage - JavaScript Arrays

adding and removing items at either end of an array

- a few simple methods we can use to add items to and remove items from an array:
  - *push - adds an item to the end of the array*
  - *unshift - adds an item to the beginning of the array (existing items are moved forward one index posn)*
  - *pop - removes an item from the end of the array*
  - *shift - removes an item from the beginning of the array (existing items are moved back one index posn)*
- **n.b.** push and pop are faster than shift and unshift due mods of the index…

Practical usage - JavaScript Arrays

adding and removing items at any array location

- if we simply delete an array item
  - *we leave a hole at that index position with `undefined`…*
  - *array length will still include this hole…*
- instead we need to use the `splice` method for insertion and deletion
- e.g.

```
var removedItems = readers.splice(1, 1);
```

- this removes a single item at index posn 1
- `splice` method will also return its own array of deleted items.
- using `splice` method
  - *also insert items into arbitrary positions in an array*
- e.g. consider the following code:

```
removedItems = readers.splice(1, 2, "cat", "rose", "violet");
//readers: ["daisy", "cat", "rose", "violet"]
//removedItems: ["emma", "yvaine"]
```

- starting from index 1
  - *it first removes two items*
  - *then adds three items: "Mochizuki", "Yoshi", and "Momochi"*
  - *algorithm defined and working…*

Practical usage - JavaScript Arrays

common operations on arrays

- some common operations on JS arrays include,
  - *iterate - traverse arrays*
  - *map - map existing array items to create a new array based on these items*
  - *test - check array items match certain conditions*
  - *find - find specific array items*
  - *aggregate - compute a single value based on array items, e.g. compute total for array from array items...*

Practical usage - JavaScript Arrays

common operations on arrays - iterate with `forEach`

- all JS arrays have a built-in method for `forEach` loops
- e.g.

```
const archives = ['waldzell', 'mariafels'];

archives.forEach(archive => {
    console.log(`archive name = ${archive}`);
});
```

Practical usage - JavaScript Arrays

common operations on arrays - map arrays

- with array mapping
  - *creating a new array based on the items in an existing array*
  - *become common usage in JavaScript development*
- idea is simple
  - *we map each item from one array to a new item in a new array*
  - *we might extract just names from an array of archives*
- e.g.

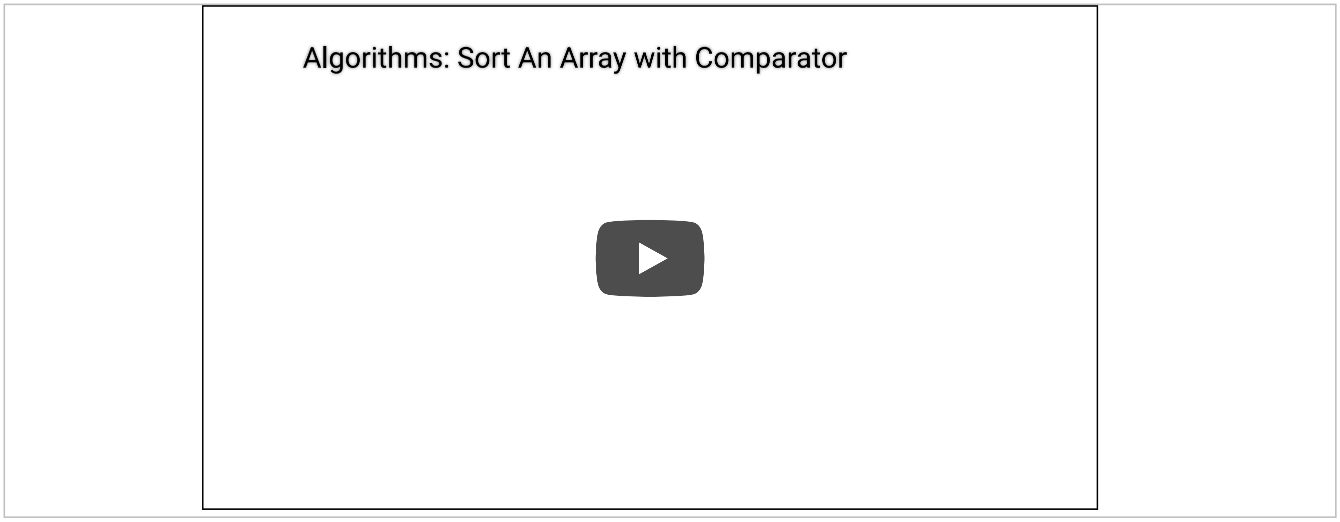
```
// array
const archives = [
  {name: 'waldzell', type: 'game'},
  {name: 'mariafels', type: 'benedictine'}
];

// map array items to new array
const archiveNames = archives.map(archive => archive.name);

// iterate through new array
archiveNames.forEach(archive => {
  console.log(`archive name = ${archive}`);
});
```

Video - Fun example

Java - comparator array sort



Array Usage - Sort an Array with Comparator - Java

Source - Sort an Array with Comparator - YouTube

Resources

JavaScript

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

Java

- Linked Lists - YouTube
- Sort an Array with Comparator - YouTube

Python

- Iterators vs Iterables - YouTube
- Linked Lists - YouTube
- Lists vs Arrays, NumPy - YouTube

Various

- Big O Notation - YouTube
- Process Manager - YouTube
- Process Manager - Algorithms and Management - YouTube
- Process Manager - Scheduler - YouTube
- Text based game Vexed with linked list - YouTube