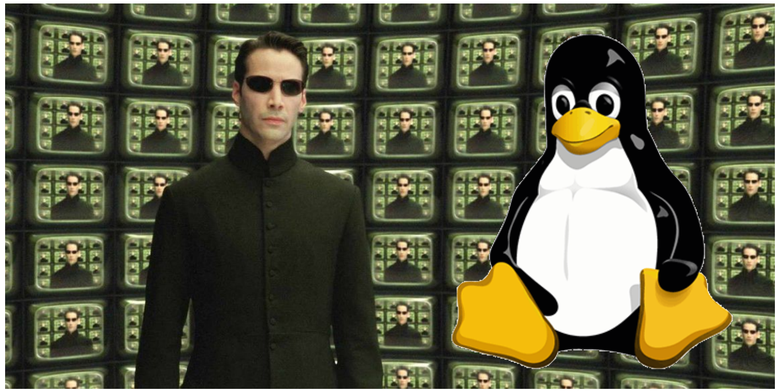
29.07.20 12:22

CSCI 125
Introduction to Computer Science and Programming II
Lecture 7: Data Structure II



Jetic Gū

2020 Summer Semester (S2)

Some changes

- Assignment 4 and Lab 4 will be released this week, covering Lecture 7
- No more Assignment 5 and Lab 5: less work for you
- Last batch of OJ problem: 9 problems p025-p033

Admin

Overview

- Focus: Data Structures
- Architecture: Linux/Unix OS
- Core Ideas:
    1. Algorithm Analysis
    2. Queue

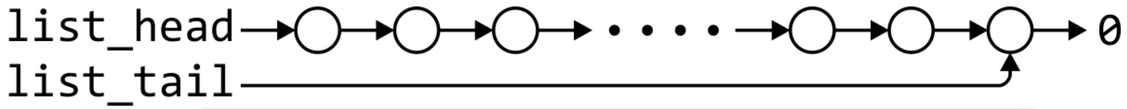
Algorithm Analysis

Complexity

Summary

Linked-List Implementation

- Operations at the front of a singly linked list are all very efficient: constant steps



	Front/1^{st}	Back/n^{th}
Find	constant	constant
Insert	constant	constant
Erase	constant	up to n

Estimation of performance

- The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

Review

Algorithm Analysis

- Predict performance
- Compare different data structures and performance
  - The performance of different **operations** of data structures
  - The performance of different **implementations** of data structures
    - e.g. Stack: Array implementation vs. linked list implementation
- Avoid major problems in performance in production

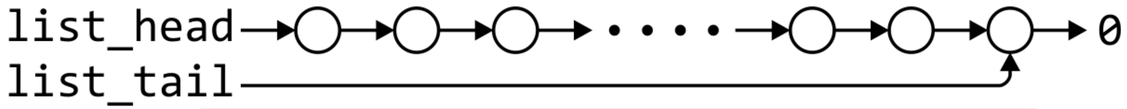
Concept

Why Performance?

- Ensure efficient execution of programme
  - e.g. terrible optimisation of Microsoft Office on macOS
- Ensure efficient processing of operations
  - such as for a web server, handling user requests
- Algorithm Analysis
  - Provide **guarantees**

Linked-List Implementation

- Operations at the front of a singly linked list are all very efficient: constant steps



	Front/1^{st}	Back/n^{th}
Find	constant	constant
Insert	constant	constant
Erase	constant	up to n

Estimation of performance

- The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

Review

Problem!

	Front/1^{st}	Back/n^{th}
Find	constant	constant
Insert	constant	constant
Erase	constant	up to n

- **up to n** is hardly a scientific expression
  - Actual implementation can take much more
    - Addition of 12 and 35: ~10 steps of machine code
  - This is actually closer to some
    - $an + b$ steps, where a and b are constants

Concept

Θ notation

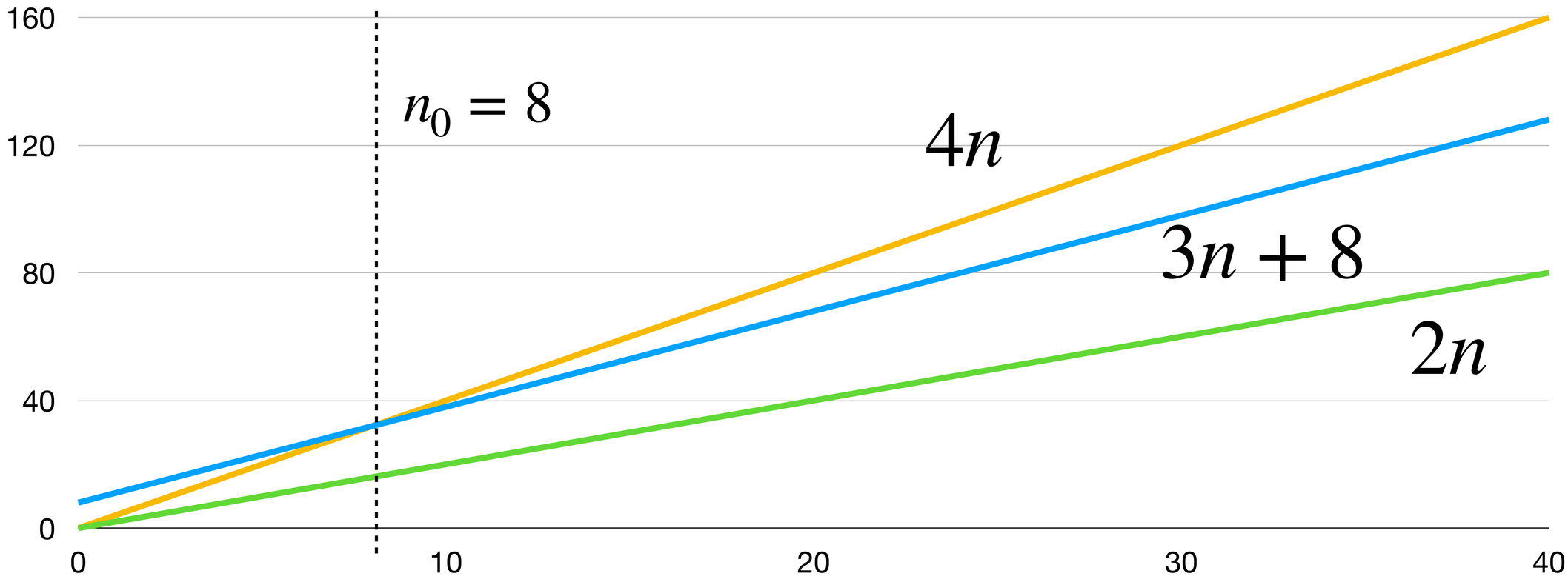
- **Θ Notation**
  - $f(n) = \Theta(g(n))$, where $g(n)$ is an **asymptotically** tight bound for $f(n)$
  - **asymptotically** tight bound: there exists constants c_1, c_2, and n_0, such that for all $n > n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Concept

Θ notation

- Example: $3n + 8 = \Theta(n)$



Degree of Change

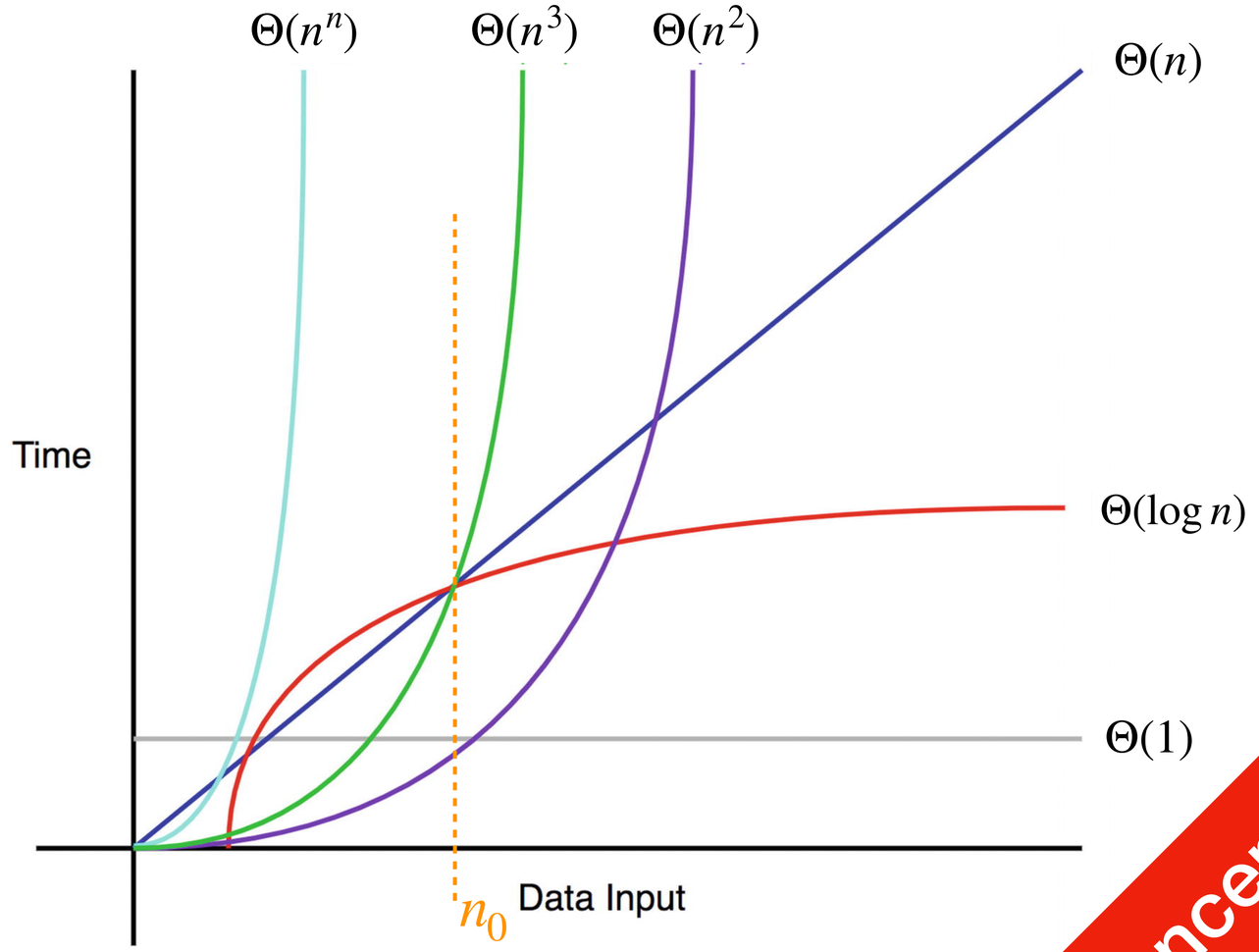
- Constant: $\Theta(1)$
  actual steps: c, where c is a constant
- Linear: $\Theta(n)$
  actual steps: $c_1 n + c_0$, where c_i are constants
- Quadratic: $\Theta(n^2)$
  actual steps: $c_2 n^2 + c_1 n + c_0$, where c_i are constants
- ....

Concept

Degree of Change

- Say you are using Θ to describe time complexity for an operation with n as input size

- e.g. $\Theta(n) > \Theta(\log n) > \Theta(1)$
  There is always an n_0, after which the higher degrees of change will make the operation more costly



Concept

Big-O notation

- **Θ Notation:** $f(n) = \Theta(g(n))$
  - **asymptotically** tight bound: there exists constants c_1, c_2, and n_0, such that for all $n > n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

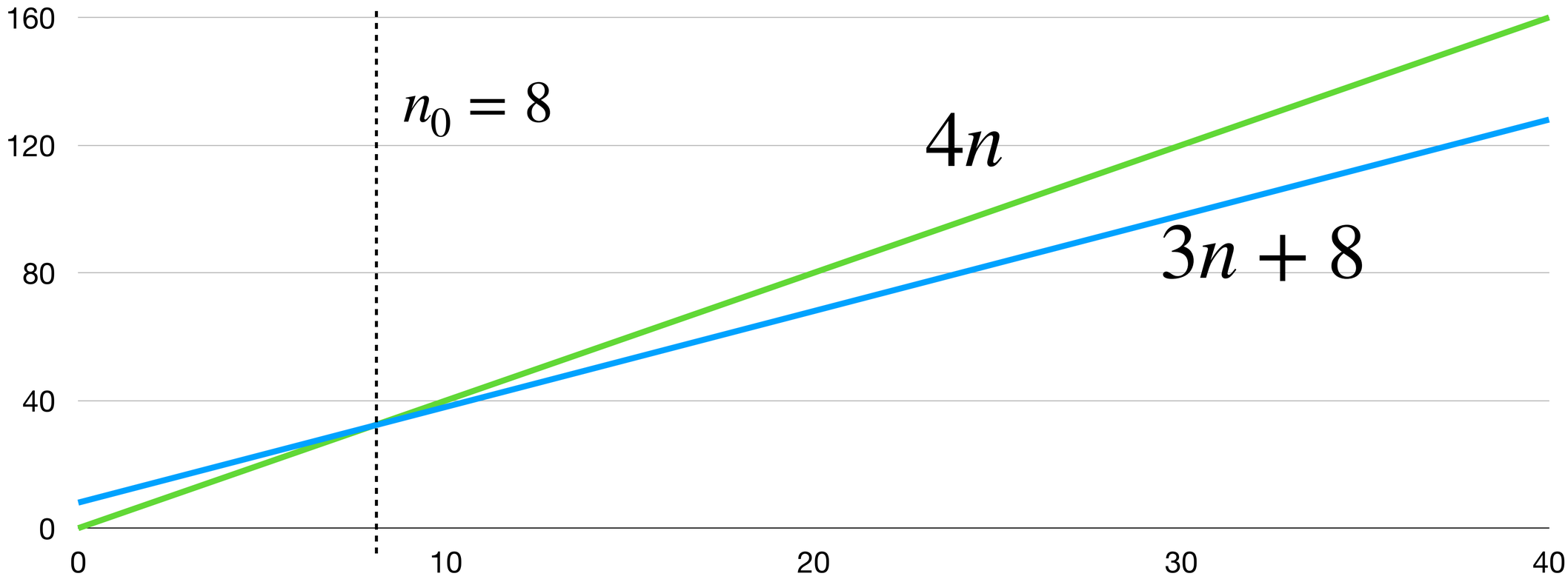
- **Big-O Notation:** $f(n) = O(g(n))$**, where** $g(n)$ **is an asymptotical upper bound for** $f(n)$
  - **asymptotical upper bound**: there exists constants c, and n_0, such that for all $n > n_0$,

$$f(n) \leq cg(n)$$

Concept

Big-O notation

- Example: $3n + 8 = O(n)$



Big-O notation

- Why Big-O?
  - If an algorithm is $\Theta(g(n))$, it is $O(g(n))$
    But **NOT** the other way, since $O(g(n))$ **only gives you the upper bound**
  - Sometimes calculating Θ is not practical, but Big-O tells us the **worst-case** scenario
  - Θ also gives us the lower-bound, which we do not usually care about

Concept

Degree of Change

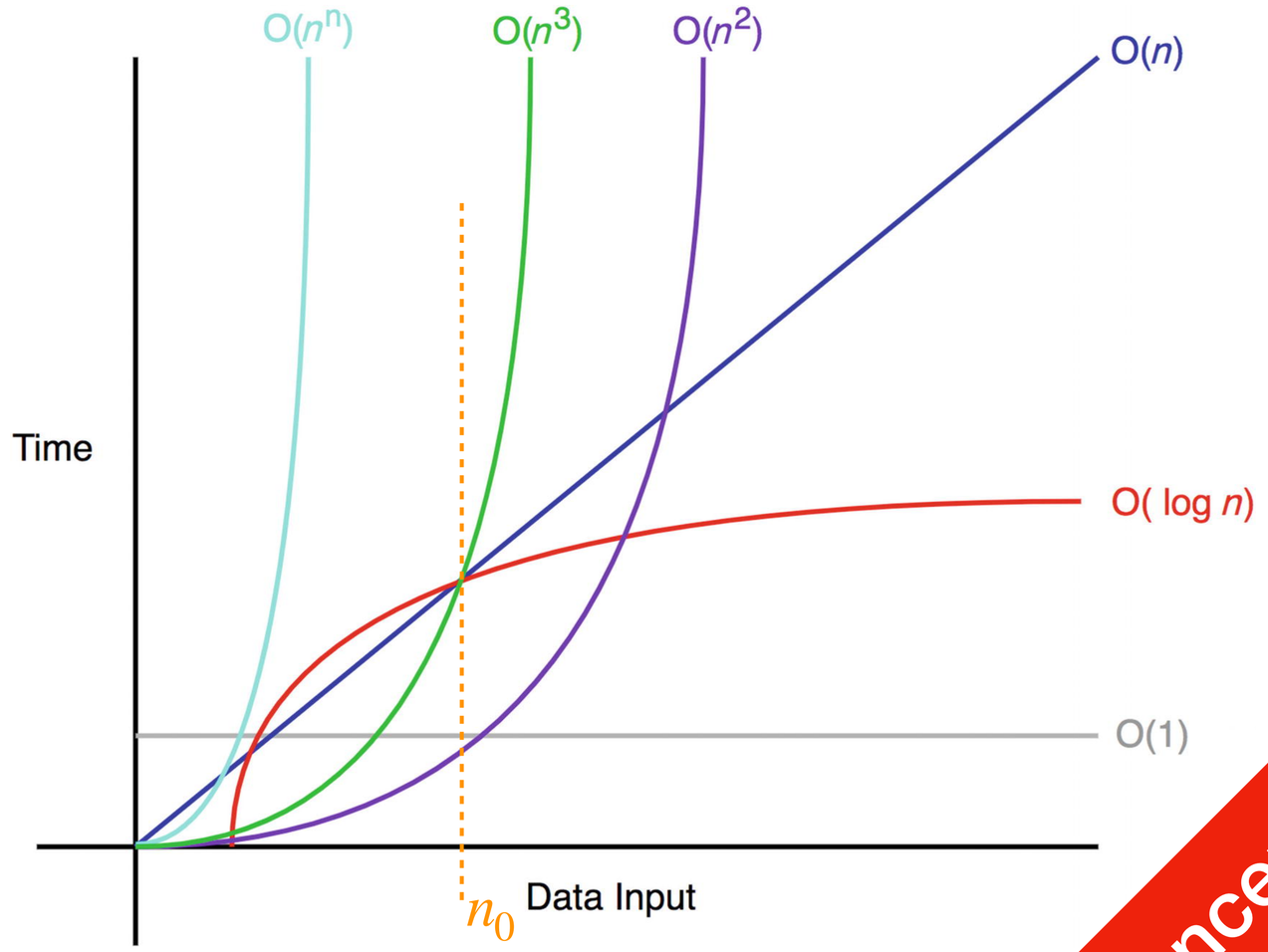
- Constant: $O(1)$
  e.g. perform a single addition
- Linear: $\Theta(n)$
  e.g. sum up n numbers
- Quadratic: $\Theta(n^2)$
  e.g. selection sort, bubble sort
- ....

Concept

Degree of Change

- Say you are using Big-O to describe time complexity for an operation with n as input size

- e.g. $O(n) > O(\log n) > O(1)$
  There is always an n_0, after which the higher degrees of change will make the operation more costly

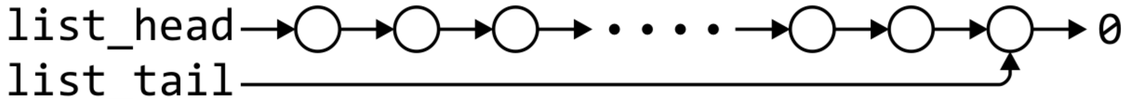


Concept

P1
Complexity

Linked-List Implementation

- Operations at the front of a singly linked list are all very efficient: constant steps



	Front/1^{st}	Back/n^{th}
Find	constant $O(1)$	constant $O(1)$
Insert	constant $O(1)$	constant $O(1)$
Erase	constant $O(1)$	up to $O(n)$

- The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

Concept

Complexities

- Time complexity
  - **Estimation** of time it takes to execute...
  - 1 second on modern 3.0GHz CPU: **roughly** 10^9 steps, e.g. $O(n^3), n \leq 1000$; $O(n^2), n \leq 10000$;
  - **Sometimes constants do vary**
- Space complexity
  - Estimation of memory space / storage space it takes

Concept

Example: Bubble Sort 1

```
func BubbleSort(arr):
    for i from 1 to n:
        for j from 1 to n-1:
            if (arr[j]> arr[j-1]):
                swap(arr[j], arr[j-1])
```

- Time complexity: $\Theta(n^2)$; Spatial Complexity: $\Theta(n)$

Exercise

Example: Bubble Sort 2

```
func BubbleSort(arr):
    sorted = false;
    while sorted == false:
        sorted = true;
        for j from 1 to n-1:
            if (arr[j]> arr[j-1]):
                swap(arr[j], arr[j-1]);
                sorted = false;
```

`sorted` is detecting whether the array is already sorted

In this case the lower-bound can be linear (already sorted), so we can only use Big-O for worse-case scenario

- Time complexity: $O(n^2)$; Spatial Complexity: $\Theta(n)$

Exercise

Example: MyList

- What are the complexities of each of these operations?
  - `prepend(...)`
  - `append(...)`
  - `get(0), get(n/2), get(n)`
  - `delete(0), delete(n/2), delete(n)`

```
1. class MyList {
2.     int value;
3.     MyList* next;
4.     MyList* last;
5. public:
6.     int length;
7.     MyList();
8.     ~MyList();
9.     void prepend(int val);
10.    void append(int val);
11.    int get(int ind);
12.    int give(int ind, int val);
13.    int delete(int ind);
14.};
```

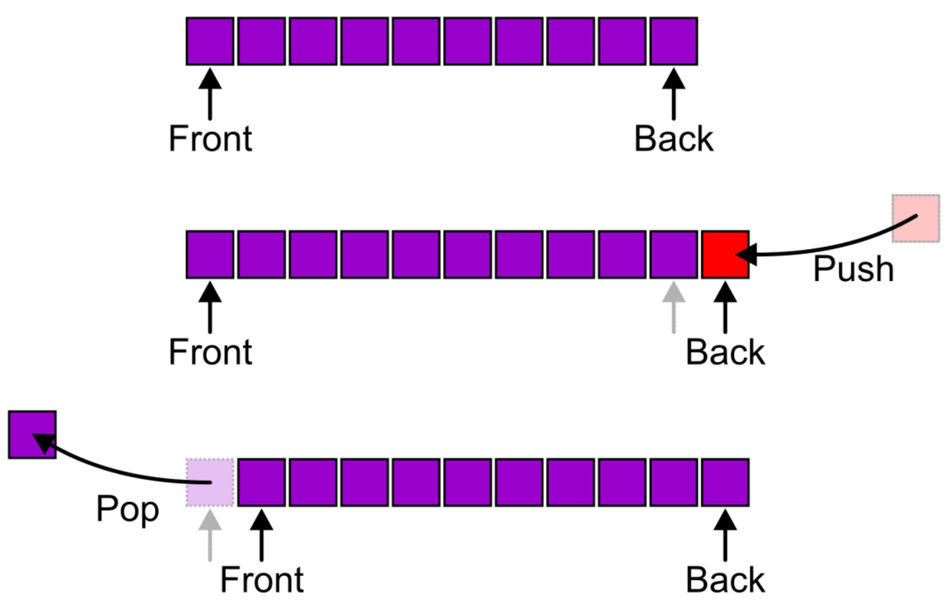
Think

Queue

Summary

Queue

- Queue: first-in–first-out (FIFO)
  - Front: return frontal element
  - Pop: remove frontal element, return its value
  - Push: insert element to the back



Concept

Queue

- Queue: first-in–first-out (FIFO)
  - Front: return frontal element
  - Pop: remove frontal element, return its value
  - **Push: insert element to the back**
- Stack: last-in–first-out (LIFO)
  - Top: return top element
  - Pop: remove top element, return its value
  - **Push: insert element to the Top**

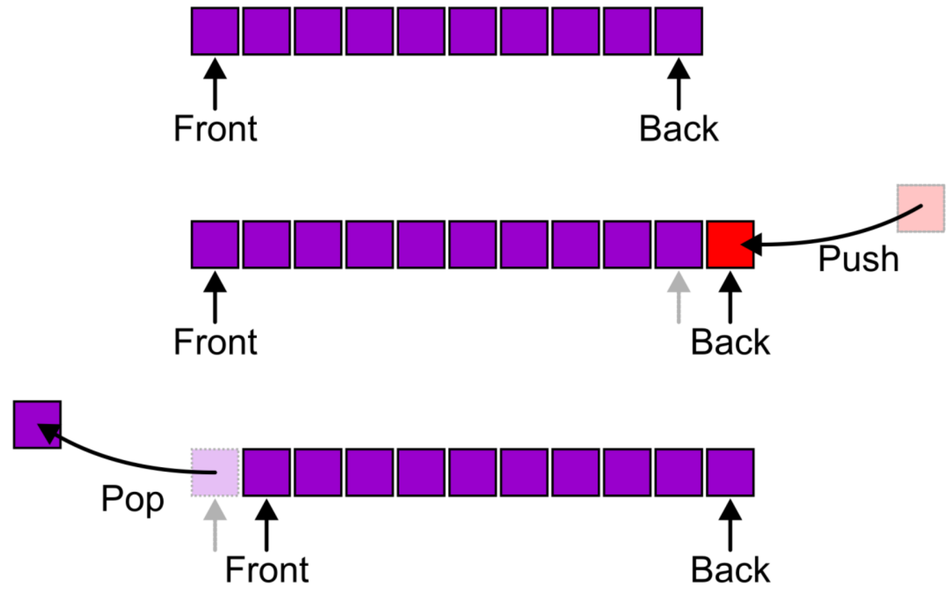
Concept

Queue

- Exceptions

  1. Calling **Pop** on an empty queue

  2. Calling F**ront** on an empty queue



Concept

Applications of Queue

- Webserver request processing
  - Multiple clients may be requesting services from one or more servers
  - Some clients may have to wait while the servers are busy
  - Those clients are placed in a queue and serviced in the order of arrival
- Most shared computer services are servers:
  - Web, file, ftp, database, mail, printers, ssh, WOW, *etc*.

Concept

Implementations

- Implementations of queues:
  - Singly linked lists (`MyList`)
  - Why is it not feasible to use a standard array?
- Requirements
  - All queue operations must run in $\Theta(1)$ time

Concept

Queue-as-List Class

- The `Queue` class using a singly linked list has a single private member variable
- 5: `empty`
  returns whether the stack is empty, $\Theta(1)$
- 6: `front`
  returns the frontal element $\Theta(1)$
- 7: `push`
  insert new element **to the back** $\Theta(1)$
- 8: `pop`
  remove frontal element, return its value $\Theta(1)$

```
1. class Queue {
2.   private:
3.     MyList list;
4.   public:
5.     bool empty();
6.     int front();
7.     void push(int val);
8.     int pop();
9. };
```

Concept

Queue-as-List Class

- The `empty` and `push` functions just call the appropriate functions of the `MyList` class

- Similar to our `Stack` implementation, but on line 5 we have `append` instead of `prepend`

```
1. bool Queue::empty() {
2.    return list.length==0;
3. }

4. void Queue::push(int val) {
5.    list.append(val);
6. }
```

Concept

Queue-as-List Class

- The `front` and `pop` functions, however, must check the boundary case
- Just like in `Stack`

```
1. int Queue::front() {
2.     if ( empty() )
3.         return -1;
4.     return list.get(0);
5. }

6. int Queue::pop() {
7.     if ( empty() )
8.         return -1;
9.     return list.delete(0);
10.}
```

Concept

Implementation Using Array

For one-ended arrays, all operations at the back are constant

	Front/1^{st}	Back/n^{th}
Find	O(1)	O(1)
Insert	O(n)	O(1)
Erase	O(n)	O(1)

Concept

Implementation Using Array

Using a two-ended array[1], $O(1)$ are possible by pushing at the back and popping from the front

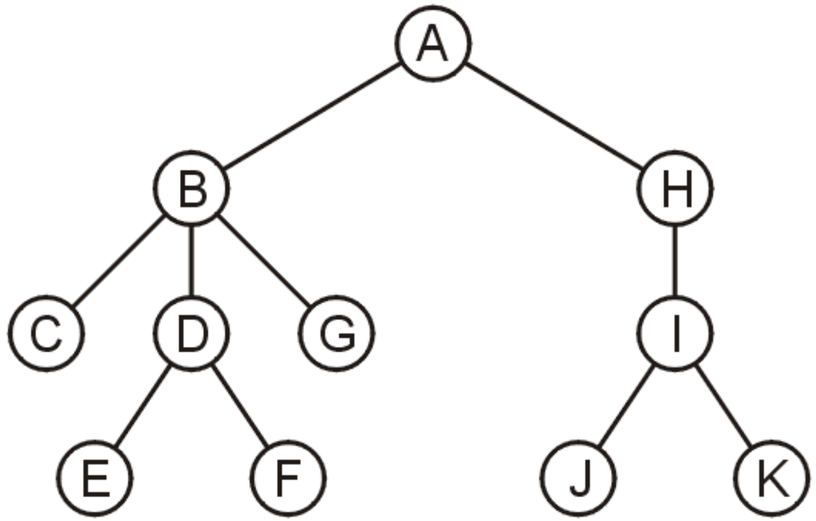
	Front/1^{st}	Back/n^{th}
Find	O(1)	O(1)
Insert	**O(1)**	O(1)
Erase	**O(1)**	O(1)

Concept

1. Not required

Application

- Traversal of a directory tree
  - Consider searching the directory structure

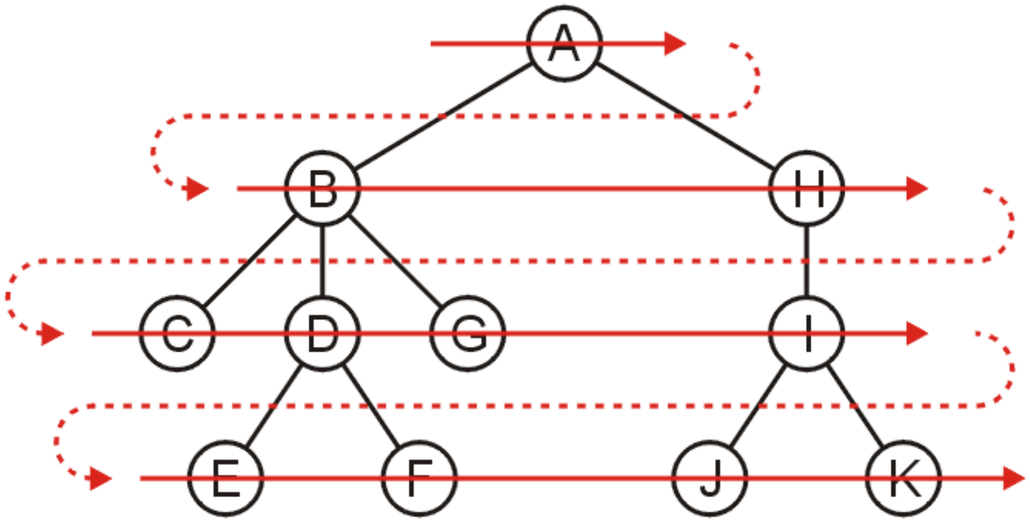


Demo

P2
Queue

Tree BFS

- Breadth-First Search (BFS)
  - Search all the directories at one level before descending a level
  - Classic searching algorithm



Demo

P2
Queue

Tree BFS

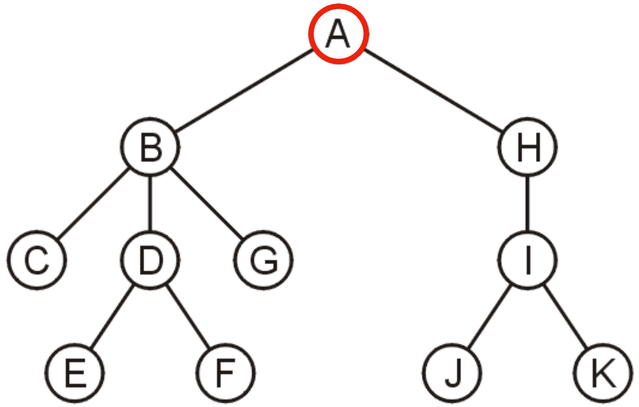
- The easiest implementation is:
  - Place the root directory into a queue
  - While the queue is not empty:
    - Pop the directory at the front of the queue
    - Push all of its sub-directories into the queue
- The order in which the directories come out of the queue will be in breadth-first order

Demo

P2
Queue

Tree BFS

Push the root directory A



Visited

Not yet

x Current

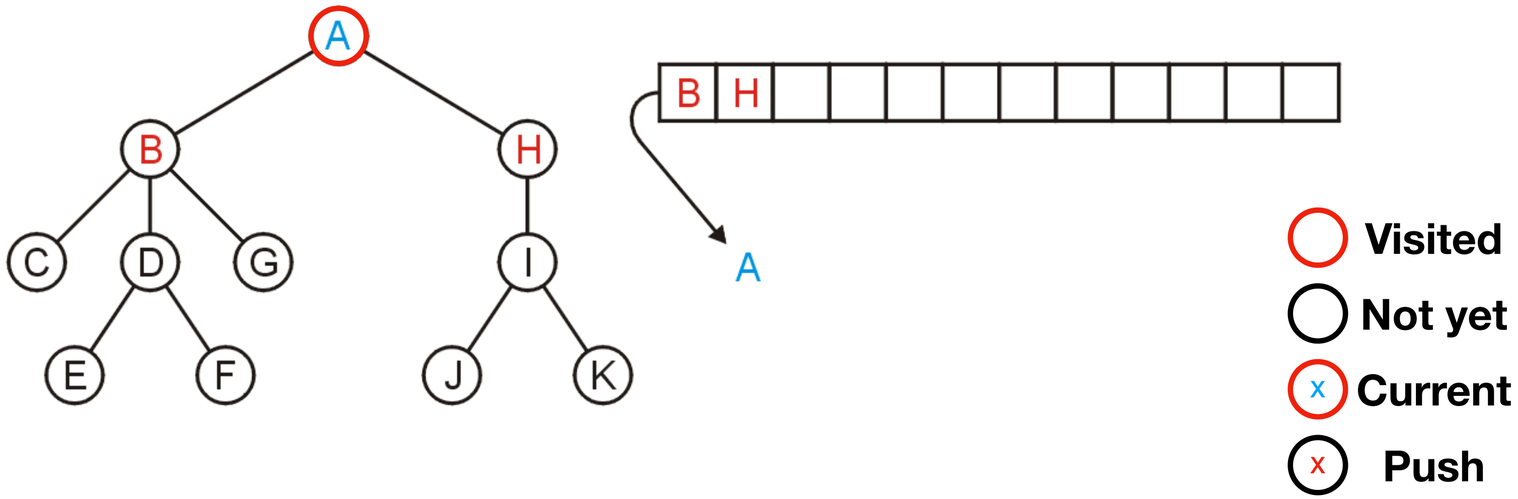
x Push

Demo

P2
Queue

Tree BFS

Pop A and push its two sub-directories: B and H

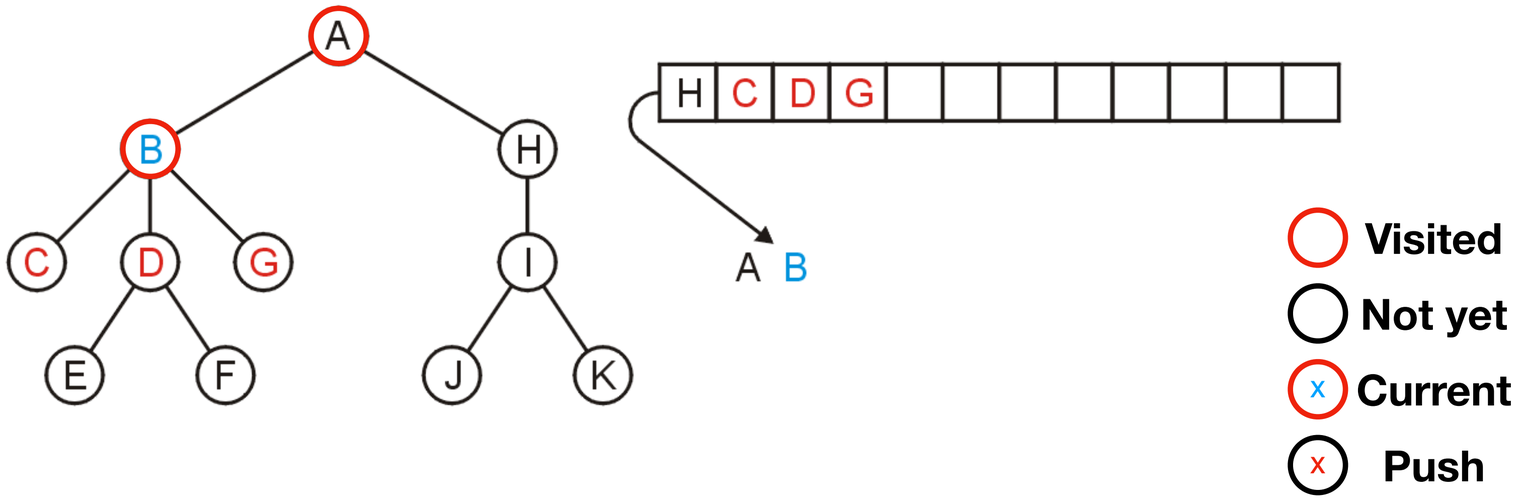


Demo

P2
Queue

Tree BFS

Pop B and push C, D, and G

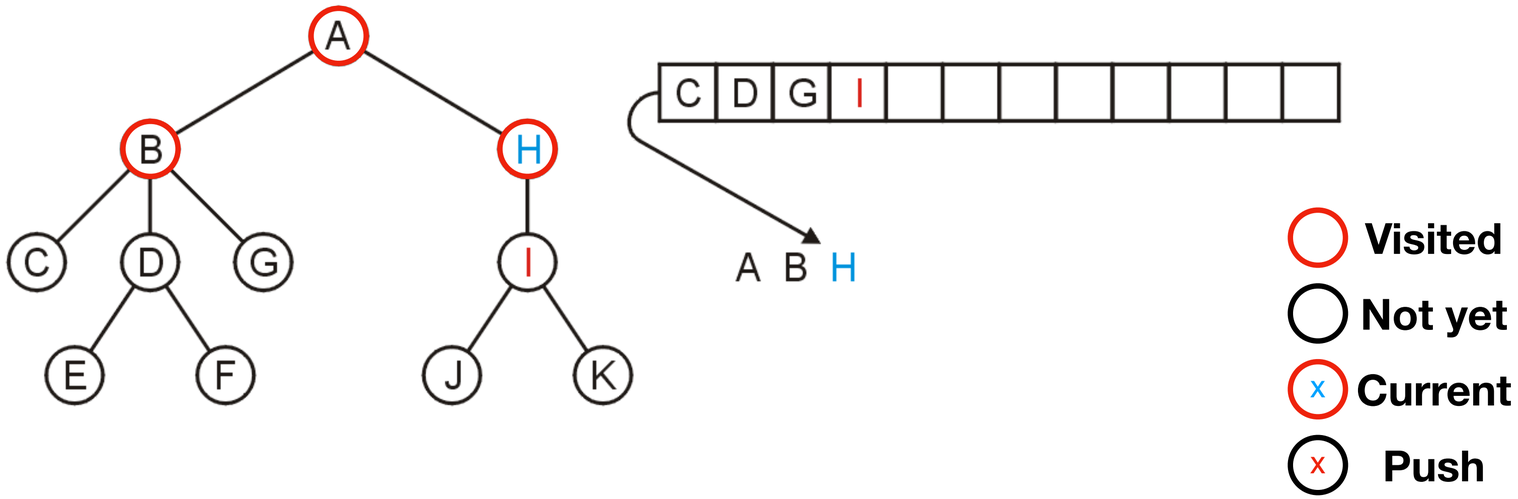


Demo

P2
Queue

Tree BFS

Pop H and push its one sub-directory I

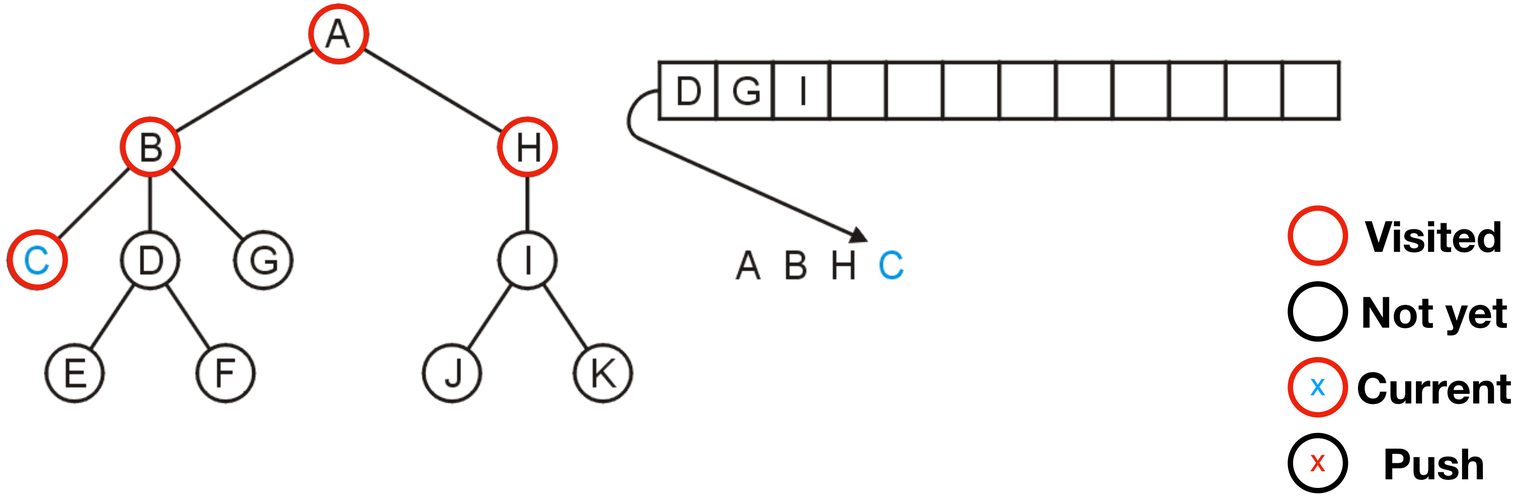


Demo

P2
Queue

Tree BFS

Pop C: no sub-directories

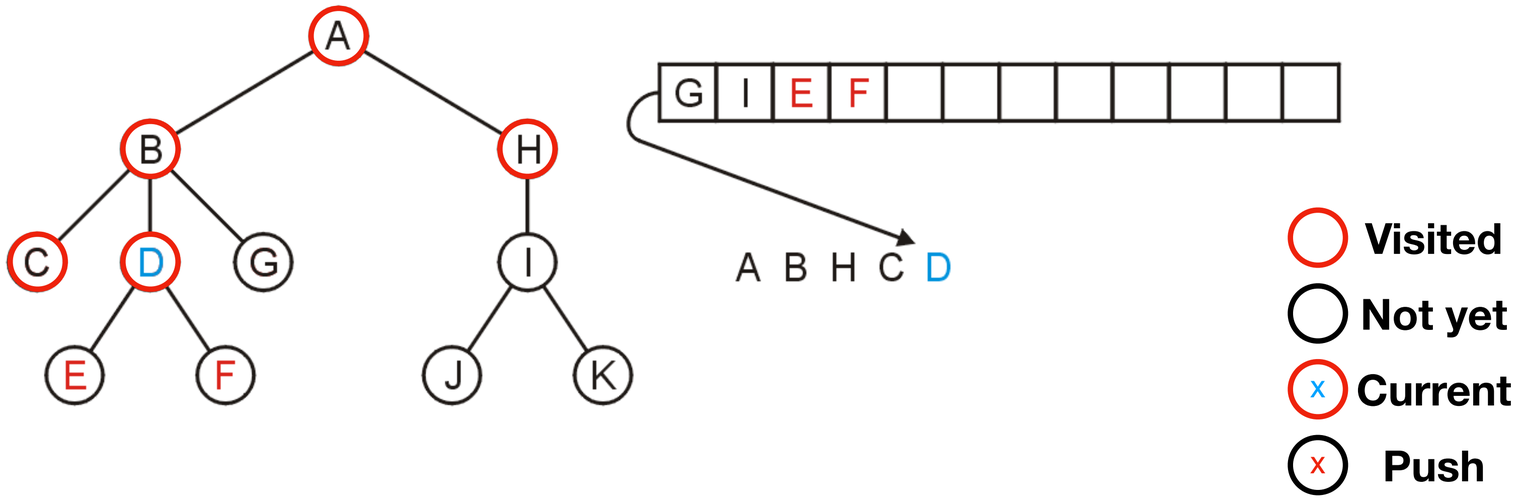


Demo

P2
Queue

Tree BFS

Pop D and push E and F

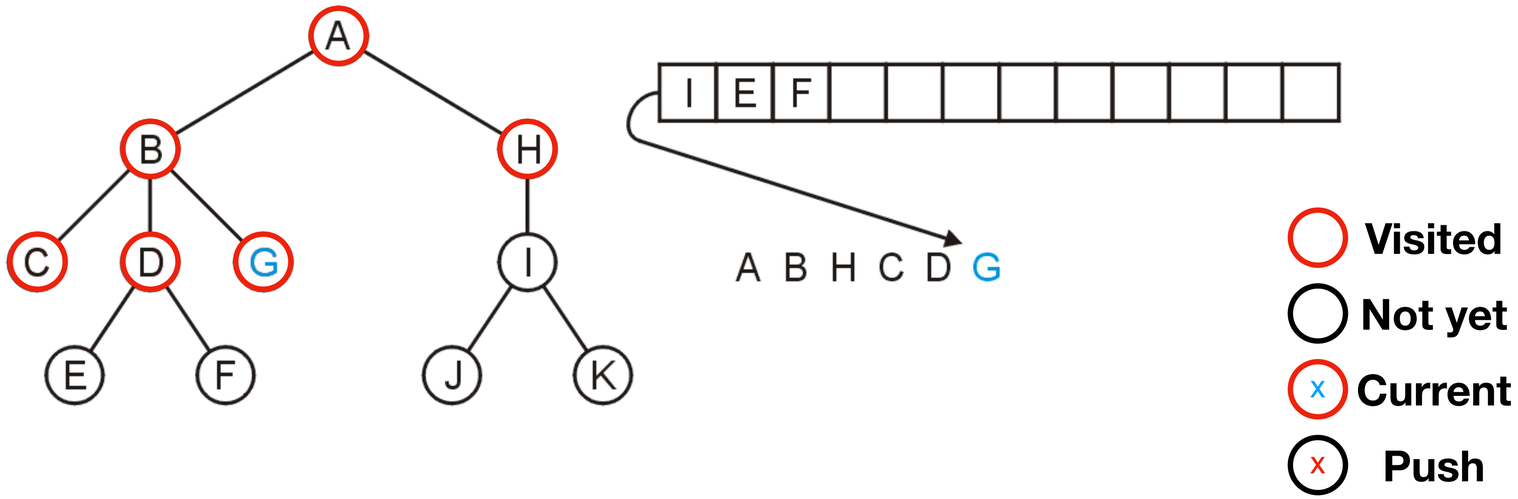


Demo

P2
Queue

Tree BFS

Pop G

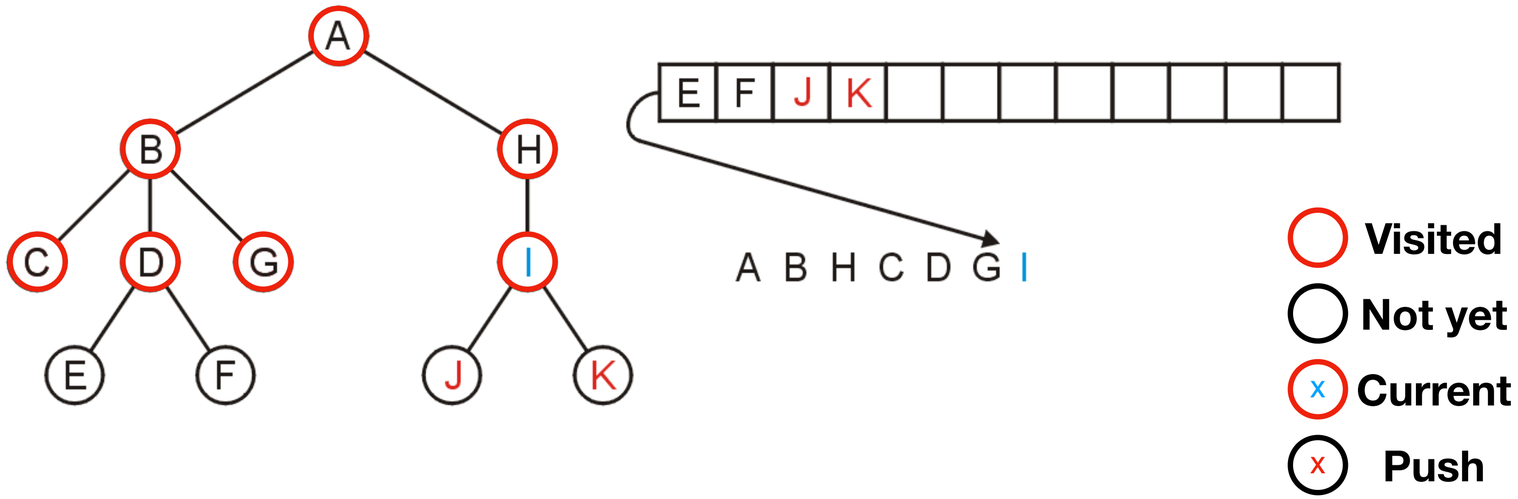


Demo

P2
Queue

Tree BFS

Pop I and push J and K

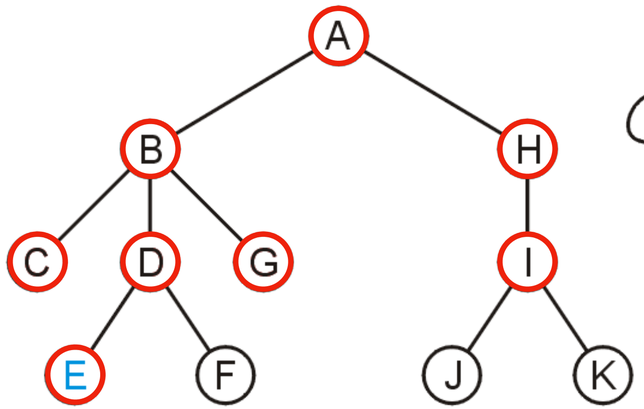


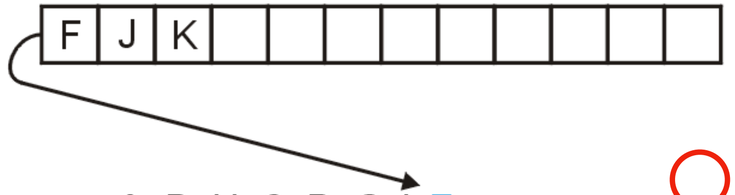
Demo

P2
Queue

Tree BFS

Pop E





Visited

Not yet

x Current

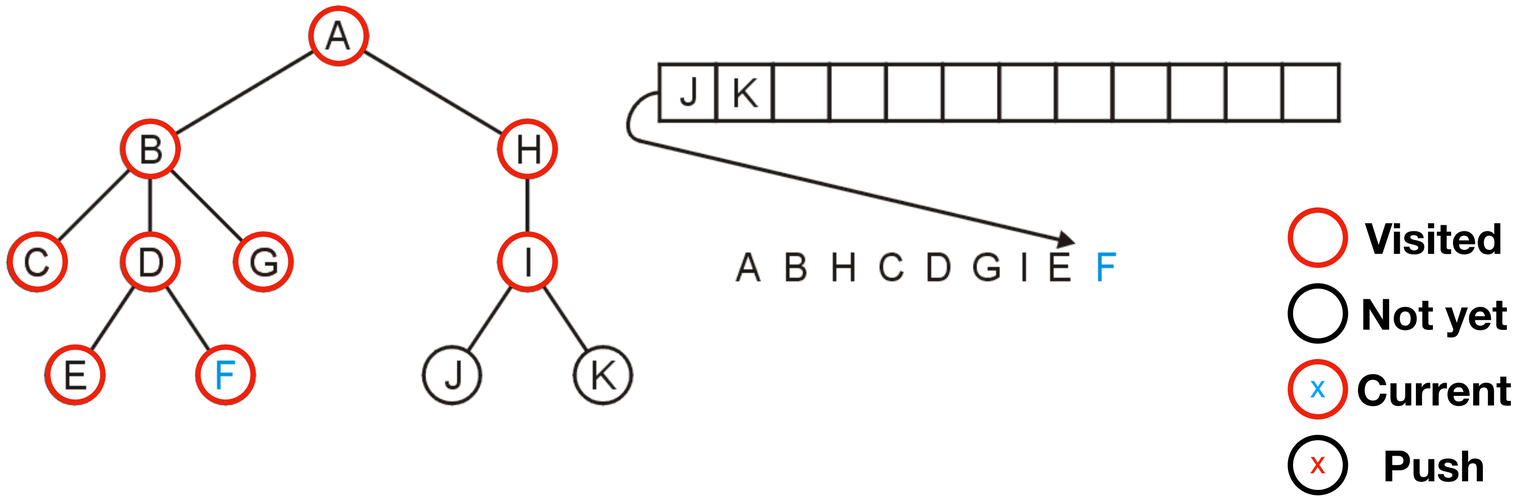
x Push

Demo

P2
Queue

Tree BFS

Pop F

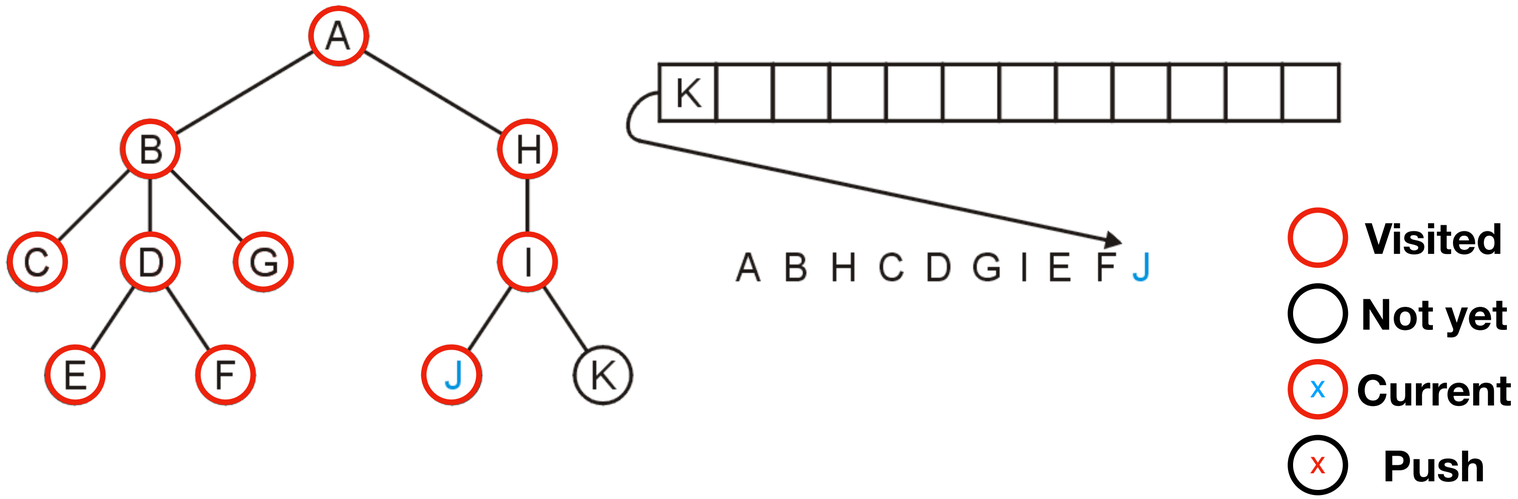


Demo

P2
Queue

Tree BFS

Pop J

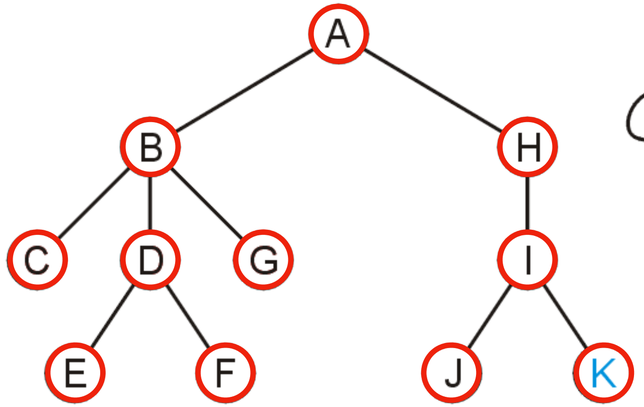


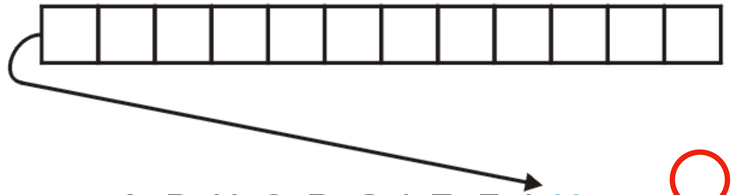
Demo

P2
Queue

Tree BFS

Pop K and the queue is empty





Visited

Not yet

x Current

x Push

Demo

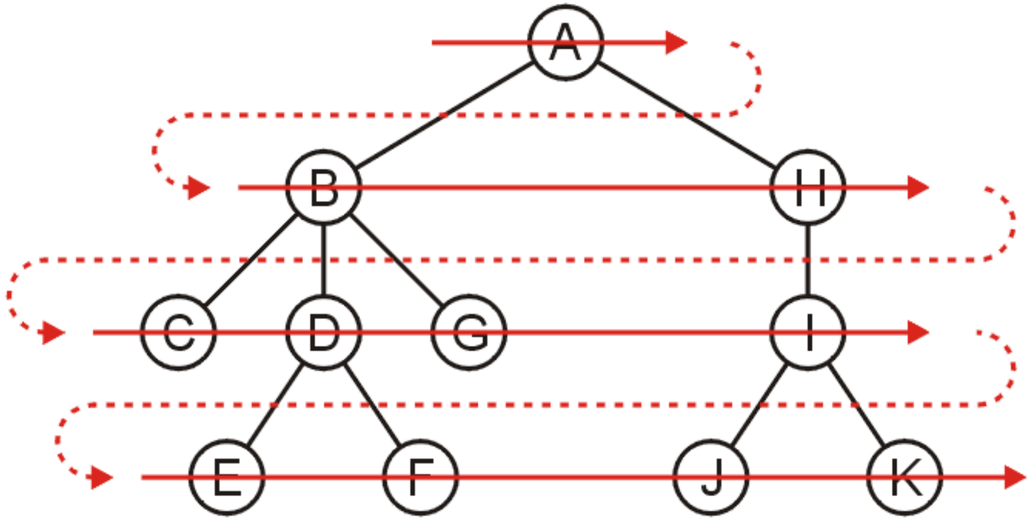
Application

- The resulting order

  `A-B-H-C-D-G-I-E-F-J-K`

  is in breadth-first order

- What is the time and space complexity



Demo

Summary

- Lecture 7: Data Structure 1-3
  - Linked List
  - Stack
  - Queue
  - Θ and Big-O Notation

Future