

IADS - Revision Semester 1

Antonio León Villares

May 2021

Contents

1	Week 2 - Asymptotics	4
1.1	Basics of Asymptotics	4
1.2	Little o	4
1.3	Little omega	5
1.4	Big O	5
1.5	Big Omega	6
1.6	Big Theta	6
1.7	Asymptotics Summary	6
2	Week 3 - Sorting Algorithms and Asymptotic Analysis	7
2.1	Analysing Algorithms	7
2.2	Insert Sort	7
2.3	Merge Sort	10
2.4	Overall Runtime Comparison: Insert vs Merge	12
2.5	Space Complexity Comparison: Insert vs Merge	12
2.6	Insert vs Merge Summary	12
3	Week 3 - Program Data in Memory	13
3.1	Memory Organisation	13
3.2	Equality Testing	14
4	Week 4 - Classic Data Structures	15
4.1	Lists	15
4.1.1	Lists as Fixed-Size Arrays	15
4.1.2	Lists as Extensible Arrays	15
4.1.3	Lists as Linked Lists	17
4.1.4	Summary of List Implementations	17
4.2	Stacks	18
4.2.1	Actions on Stacks	18
4.2.2	Stack Implementation: Array vs Linked List	18
4.3	Queues	19
4.3.1	Actions on Queues	19
4.3.2	Queues as Wraparound Arrays	20

4.3.3	Queues as Linked Lists	20
4.3.4	Comparison of Queue Implementations	21
4.4	Sets and Dictionaries	21
4.5	Hashing	22
4.5.1	Dealing with Clashes: Hash Buckets	22
4.5.2	Dealing with Clashes: Open Addressing and Probing . . .	23
5	Week 5 - Balanced Trees	24
5.1	Motivation	24
5.2	Binary Trees	24
5.2.1	Contains	25
5.2.2	Insert	26
5.2.3	Delete	27
5.2.4	Binary Trees Summary	27
5.3	Red-Black Trees	28
5.3.1	Insert and the Red-Uncle Rule	29
5.4	Summary of Balanced Trees	32
5.5	Data Structures Review	32
6	Week 5 - Divide-Conquer-Combine and the Master Theorem	33
6.1	Runtime of Recursive Algorithms	33
6.2	The Master Theorem	34
7	Week 7 - The Heap Data Structure	36
7.1	The Heap	36
7.2	Operations on Heaps	37
7.2.1	Heap-Maximum	37
7.2.2	Max-Heapify	37
7.2.3	Heap-Extract-Max	39
7.2.4	Max-Heap-Insert	39
7.2.5	Build-Max-Heap	41
7.2.6	Summary of Heap Operations	42
7.3	HeapSort	42
7.4	Priority Queues as Heaps	44
8	Week 8 - Quicksort	45
8.1	Overview	45
8.2	Quicksort Runtime Analysis	51
9	Week 9 - Graphs: Representation and Searching	53
9.1	Graphs	53
9.2	Representing Graphs	54
9.2.1	Adjacency Matrix	54
9.2.2	Adjacency List	55
9.2.3	Graph Representation Comparison	55
9.2.4	Sparse and Dense Graphs	56

9.3	Traversing Graphs	56
9.3.1	Breadth-First Search	56
9.3.2	Depth-First Search	58
9.3.3	Recursive Depth-First Search	60
9.3.4	Runtime Analysis of Traversal Strategies	60
9.4	DFS Forests	61
9.5	Topological Ordering and TopologicalSort	61
9.5.1	Topological Order	61
9.5.2	Theorems	62
9.5.3	Topological Sort	63
9.6	Connected Components	65

1 Week 2 - Asymptotics

1.1 Basics of Asymptotics

- **Asymptotic Analysis:** use of mathematical functions to make precise, quantitative statements about the efficiency of an algorithm. We use simple functions to bound the runtime behaviour of algorithms.
- **Simple Functions:**
 - *Polynomial:* $1, n, n^2, \dots$
 - *Logarithmic:* $\lg(n), n\lg(n)$
 - *Exponential:* 2^n
 - *Roots:* \sqrt{n}
 - *Weird:* 2^{2^n}
- **Requirements for Asymptotics:** consider 2 algorithms with runtime given by $T_1(n)$ and $T_2(n)$, where n is the size of the input. Then, we want:
 1. to capture that eventually one of the 2 is greater than the other
 2. to gauge how much faster an algorithm is over the other
 3. to ensure that independently of implementation detail, an algorithm will always be better than the other
 - no matter if the best programmer codes up and inefficient algorithm in the best, fastest, machine, a more efficient algorithm will eventually always fare better

1.2 Little o

- **Definition of Little o:** let $f(n)$ and $g(n)$ be functions. Then, we say that **f is $o(g)$** if f grows slower (is asymptotically smaller) than g . Informally, we expect:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$$

Formally, **f is $o(g)$** if:

$$\forall c > 0, \exists N : \forall n \geq N, f(n) < c \times g(n)$$

This is equivalent to saying:

“eventually, no matter how much we scale g , f will always be smaller than g ”

- **Meaning of Asymptotics:** asymptotics actually represent a set, so saying that **f is $o(g)$** is saying that $f \in o(g)$, where:

$$o(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \forall c > 0, \exists N : \forall n \geq N, f(n) < c \times g(n)\}$$

- **Reducing Clutter with o :** we can heavily simplify complex expressions. In particular, if **f is $o(g)$** , it is unaffected by scaling (by definition), and we can group them together. For example:

$$\begin{aligned}
& (3n + 5\sqrt{n} + 17lg(n))(4n + \frac{\sqrt{n}}{lg(n)} + 12) \\
&= (3n + o(n) + o(n))(4n + o(n) + o(n)) \\
&= (3n + o(n) + o(n))(4n + o(n) + o(n)) \\
&= (3n + o(n))(4n + o(n)) \\
&= 12n^2 + o(3n^2) + o(4n^2) + o(n^2) \\
&= 12n^2 + o(n^2)
\end{aligned}$$

1.3 Little omega

Let $f(n)$ and $g(n)$ be functions. Then, we say that **f is $\omega(g)$** if f grows faster (is asymptotically larger) than g . Informally, we expect:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Formally, **f is $\omega(g)$** if:

$$\forall C > 0, \exists N : \forall n \geq N, C \times g(n) < f(n)$$

This is equivalent to saying:

“eventually, no matter how much we scale g , f will always be larger than g ”

Indeed, there is a sort of reciprocity between o and ω :

$$f \in o(g) \iff g \in \omega(f)$$

1.4 Big O

Let $f(n)$ and $g(n)$ be functions. Then, we say that **f is $\mathcal{O}(g)$** if f grows no faster than g . Informally, we expect:

Formally, **f is $\mathcal{O}(g)$** if:

$$\exists C > 0, \exists N : \forall n \geq N, f(n) \leq C \times g(n)$$

This is equivalent to saying:

“eventually, no matter how much we scale g , f will always be at most as large as g ”

1.5 Big Omega

Let $f(n)$ and $g(n)$ be functions. Then, we say that **f is $\Omega(g)$** if f grows at least as fast as g . Informally, we expect:

Formally, **f is $\Omega(g)$** if:

$$\exists c > 0, \exists N : \forall n \geq N, c \times g(n) \leq f(n)$$

This is equivalent to saying:

“eventually, no matter how much we scale g , f will always be at least as large as g ”

Indeed, there is a sort of reciprocity between \mathcal{O} and Ω :

$$f \in \mathcal{O}(g) \iff g \in \Omega(f)$$

1.6 Big Theta

Let $f(n)$ and $g(n)$ be functions. Then, we say that **f is $\Theta(g)$** if f grows at the same rate as g . Informally, we expect:

Formally, **f is $\Theta(g)$** if:

$$\exists c_1, c_2 > 0, \exists N : \forall n \geq N, c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$$

This is equivalent to saying:

“eventually, no matter how much we scale g , f will always be at tightly bound by g ”

Indeed:

$$f \in \Theta(g) \iff f \in \mathcal{O}(g) \ \& \ f \in \Omega(g)$$

1.7 Asymptotics Summary

- $f \in o(g)$: f grows slower than g
- $f \in \omega(g)$: f grows faster than g
- $f \in \mathcal{O}(g)$: f grows at most as fast g
- $f \in \Omega(g)$: f grows at least as fast g
- $f \in \Theta(g)$: f grows at the same rate as g

2 Week 3 - Sorting Algorithms and Asymptotic Analysis

2.1 Analysing Algorithms

- **Cost Model:** a definition of how the cost of an algorithm is measured
 - runtime, memory use, disk operations
- **Runtime Cost:** time necessary to execute an algorithm for any input of size n
 - can be measured in terms of comparisons, line executions or others
- **Best, Worst, and Average Case:** we can use our asymptotics to define bounds on the best, worst and average runtime
 - $T_{worst} \in \mathcal{O}(g)$: worst case runtime is at most as bad g (just say runtime is $\mathcal{O}(g)$)
 - $T_{worst} \in \Omega(g)$: worst case runtime is at least as bad as g
 - $T_{worst} \in \Theta(g)$: worst case runtime is as bad as g
 - $T_{best} \in \mathcal{O}(g)$: best case runtime is at most as good g
 - $T_{best} \in \Omega(g)$: best case runtime is at least as good as g (just say runtime is $\Omega(g)$)
 - $T_{best} \in \Theta(g)$: best case runtime is as good as g

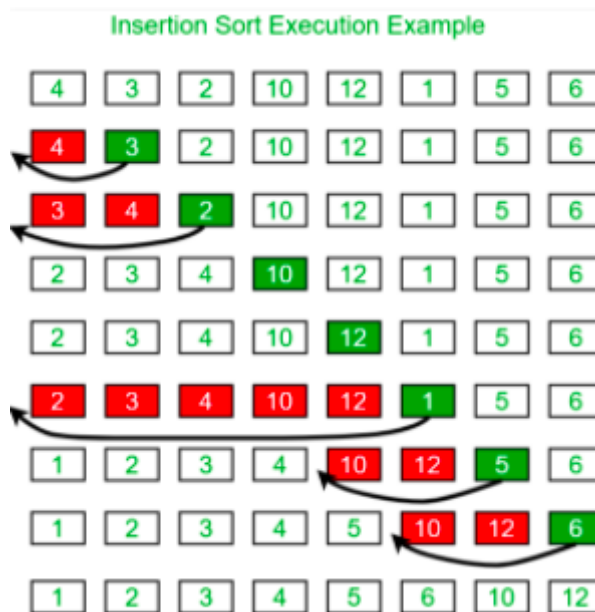
2.2 Insert Sort

```
0  InsertSort(A):
1    for i = 1 to n-1    # write n for size of A
2        x = A[i]
3        j = i-1
4        while j ≥ 0 and A[j] > x
5            A[j+1] = A[j]
6            j = j-1
7        A[j+1] = x
```

- **How it Works:** traverse through each element of the array, sorting all the elements before it.
 - get element x
 - all elements before it have been sorted (by the previous executions of the algorithm)
 - check all elements before x . If y is larger than x , then y moves one place up. Otherwise, x gets inserted after y .

- **Worked Example:** consider $[12, 11, 13, 5, 6]$

1. $i = 1$: $A[1] = 11$. Since $12 > 11$, put 12 in place of 11. Terminate while, and insert 11 before 12.
– $[11, 12, 13, 5, 6]$
2. $i = 2$: $A[2] = 13$. Since $12 \nlessdot 13$, the while loop doesn't get executed.
– $[11, 12, 13, 5, 6]$
3. $i = 3$: $A[3] = 5$. Since all elements before 5 are bigger than 5, we insert 5 at the beginning of the array.
– $[5, 11, 12, 13, 6]$
4. $i = 4$: $A[4] = 6$. Since all elements before 6 are bigger than 6, except for 5, we insert 6 before 5.
– $[5, 6, 11, 12, 13]$



- **Runtime Analysis:** most of the work occurs during the comparison steps (line 4), so we describe the runtime in terms of the number of calls.
 - for each i , the while loop might be called at most i times (j begins at $i - 1$ and potentially terminates at -1)
 - there are a total of $n - 1$ i 's to consider

- since for each i we make at most i comparisons, and there are $n - 1$ such i 's, the total number of comparisons is:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \mathcal{O}(n^2)$$

- **Worst Case:** the worst case occurs when the array is sorted in reverse order, as this will require that for each i , we make exactly i comparisons. In this worst case, it is clear that the runtime is $\Omega(n^2)$, so the worst case is $\Theta(n^2)$
- **Best Case:** the best case occurs when only 1 comparison is made (so the while loop is never entered). In this case, the total number of comparisons will be precisely $n - 1$, so $\Theta(n)$ runtime
- **Average Case:** can be shown that, in the average case, runtime is $\Theta(n^2)$
- **Insert Sort Summary:**
 - *Worst Case:* $\Theta(n^2)$ (input in reverse order)
 - *Average Case:* $\Theta(n^2)$
 - *Best Case:* $\Theta(n)$ (input already sorted)

2.3 Merge Sort

Merge (B,C):

```
    allocate D of size |B| + |C|
    i = j = 0
    for k = 0 to |D|-1
        if B[i] < C[j]      # Convention:  $\infty$  if index out of range
            D[k] = B[i], i = i+1
        else
            D[k] = C[j], j = j+1
    return D
```

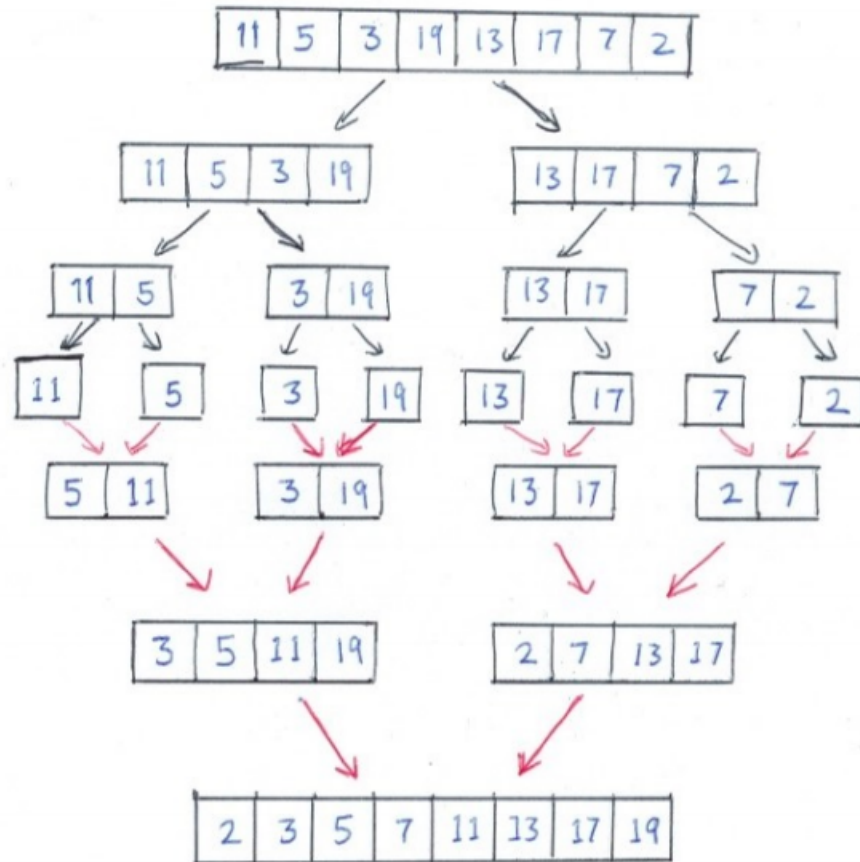
MergeSort (A,m,n): # sorts A[m], A[m+1], ..., A[n-1]
 # returning result in an array D of size n-m

```
    if n-m = 1
        return [ A(m) ]
    else
        p =  $\lfloor (m+n)/2 \rfloor$ 
        B = MergeSort (A,m,p)
        C = MergeSort (A,p,n)
        D = Merge (B,C)
        return D
```

MergeSortAll (A):

```
    return MergeSort (A,0,|A|)
```

- **How it Works:** given any array, it splits it into 2, recursively applying MergeSort on each of the halves. The returned arrays are then merged and ordered using Merge. The base case occurs when the arrays to which we apply merge sort contain a simple element, in which case they are returned.



- **Runtime Analysis:**

- **Merge:** if $m = |B| + |C|$, does at most $m - 1$ comparisons (one comparison per for loop), and at least $\min(|B|, |C|) = \frac{m}{2} - \mathcal{O}(1)$. Hence, Merge is $\Theta(m)$
- **MergeSort:** during the whole MergeSort procedure in which Merge is applied (red arrows), there are a total of n elements at each level. From before, in merging all of these elements, we will do about n comparisons (at most $n - 1$, and at least $\frac{n}{2} - \mathcal{O}(1)$). The total number of levels during which merging occurs will be (about) $\lg(n)$ (as we are doubling the length of the array at each level). Thus, at most the algorithm does $\mathcal{O}(n \lg(n))$ work, and at least $\Omega(n \lg(n))$. Thus, the general runtime of MergeSort is $\Theta(n \lg(n))$

- **Merge Sort Summary:** independent of input always does the same amount of work:

- *Worst Case*: $\Theta(n \lg(n))$
- *Average Case*: $\Theta(n \lg(n))$
- *Best Case*: $\Theta(n \lg(n))$

2.4 Overall Runtime Comparison: Insert vs Merge

We can consider overall runtime by consider that every line execution is done in $\Theta(1)$ time. Following similar thinking we get the same results:

Algorithm	Worst	Average	Best
Insert Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge Sort	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$

2.5 Space Complexity Comparison: Insert vs Merge

- **Space Complexity in Insert Sort**: since it can be done in place, its only memory requirements are in storing temporary variables, so $\Theta(1)$ space
- **Space Complexity in Merge Sort**: naively might require $\Theta(n \lg(n))$ (total size of all elements within each generated array). However, if we reclaim space occupied after we finish merging arrays, we can get $\Theta(n)$ space

2.6 Insert vs Merge Summary

Algorithm	Worst	Average	Best	Space
Insert Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
Merge Sort	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	$\Theta(n)$

- reading and writing values of variables
- accessing/updating field through dereferencing
- accessing/updating object in array
- allocating a new object in memory (`new Person(...)`)
- allocating new array to the heap

all work in constant time

3.2 Equality Testing

Two objects can be equal in 2 ways:

- have the same reference to memory (*is*)
- have the same contents (*==*)

Let:

- `L1 = [1,2,3]`
- `L2 = L1`
- `L3 = [L1,L1,L1]`
- `L4 = L1[:]`
- `L5 = L3[:]`
- `L6 = [L1[:],L1[:],L1[:]]`

Then:

- `L2 is L1` and `L2 == L1` are both *True*, as `L2` copies the reference in memory of `L1`, so then it will have the same contents and the same reference
- `L4 is L1` is *False*, as `L4` copies the contents of `L1` (so `L4 == L1` is *True*), but they are stored at a different place in the heap
- `L6 is L3` is *False*, since `L6` constructs a completely new array, albeit with the exact same elements as `L3` (so `L6 == L3` is *True*)

4 Week 4 - Classic Data Structures

4.1 Lists

- **List:** a collection of unordered items
- **Actions on a List:**
 - *get(i)*: get item at index i
 - *set(x, i)*: set item x at index i
 - *cons(x)*: insert item x at the start
 - *append(x)*: add item x at the end
 - *insert(x, i)*: add item x at index i
 - *delete(i)*: delete item at index i
 - *length*: number of items

4.1.1 Lists as Fixed-Size Arrays

- **Fixed-Size Arrays:** array which can store at most m elements
 - keeps track of current number of occupied entries with a number n (gets updated whenever we *insert/append*)
 - $\Theta(1)$: *length*, *get*, *set*, *append*
 - $\Theta(n)$: *cons*, *insert*, *delete* [**WORST CASE**]
- **Benefits and Weaknesses:**
 - ✓ fast *get* and *set* (keep array in stack)
 - ✓ fixed size \implies good memory management (space reclaimed if in stack)
 - ✗ can't cope with lists with *length* $> m$
 - ✗ many lists of unpredictable sizes \implies under-cater or over-cater
 - **Conclusion:** bad choice for general list

4.1.2 Lists as Extensible Arrays

- **Extensible Array:** if adding items to the array would cause it to overflow, create a new, bigger array to store items
 - cheap if there is space in contiguous memory
 - otherwise, need to generate a completely new array of length $length \times r$, $r \in \mathbb{R}_{>1}$, before appending the new element
 - normal append is $\Theta(1)$, but if it requires expansion, becomes $\Theta(n)$

- **Amortized Cost Analysis:** used to see behaviour of extensible arrays over a long run of appends. We want to know whether the repeated copying is cost-effective *on average*.

- array size changes: a, ar, ar^2, ar^3, \dots
- size of array after n expansions: ar^n
- so to make m appends, require $\log_r(\frac{m}{a})$ total expansions
- since for each expansion we need to copy all items from one array into the new one, the number of copyings can then be found via a geometric series. Letting $s = \lceil \log_r(\frac{m}{a}) \rceil$

$$\# \text{ copyings} = a \times \sum_{i=0}^s r^i = a \times \frac{r^s - 1}{r - 1}$$

Now, notice that we must have:

$$ar^{s-1} < m \leq ar^s$$

So:

$$a \times (r^s - 1) = ar^s - a < ar^s = (ar^{s-1})r < mr$$

Thus:

$$\# \text{ copyings} < \frac{mr}{r - 1}$$

Since we copy at most m items, the number of copyings/appends **per item** is at most:

$$\frac{r}{r - 1}$$

which is a real number

- thus, the amortized cost of append is $\mathcal{O}(1)$ per operation

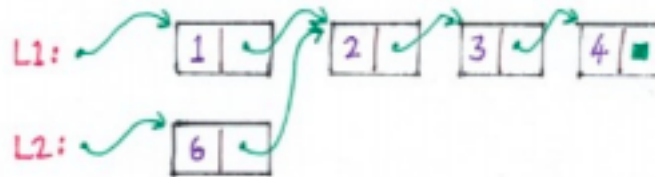
```
append (x):
  if n = |A|
    B = new array ([ n × r ])
    copy contents of A into B (n items)
    A = B
  # Now do ordinary append:
  A[n] = x
  n = n+1
```

- **Extensible Arrays Operation Costs:**

- $\Theta(n)$: *cons*, *insert*, *delete* (even amortized)
- *append*: $\Theta(1)$ best case, $\Theta(n)$ worst case, $\mathcal{O}(1)$ amortized cost

4.1.3 Lists as Linked Lists

- **Linked List:** has nodes which contain a key (value of item) and a reference to the next item in the list
 - $\Theta(n)$: *get*, *set*, *insert*, *delete* [**WORST CASE**]
 - * *insert* and *delete* can be $\Theta(1)$ if we want to insert at cell i , and we already have location of cell $i - 1$
 - $\Theta(1)$: *cons*
 - we can also share different parts of a list between linked lists, as we only need references



4.1.4 Summary of List Implementations

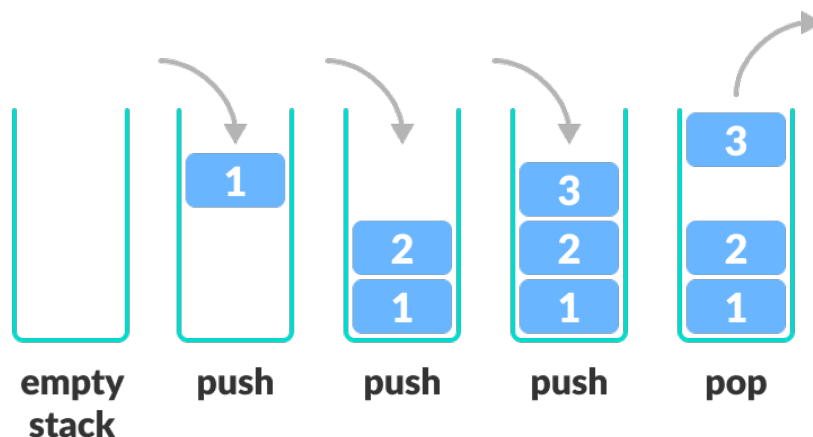
We consider upper bounds on runtime for the list implementations:

Operation	Array impl	Linked-list impl
get	$O(1)$	$O(n)$
set	$O(1)$	$O(n)$
cons	* $O(n)$	$O(1)$
append	* $O(n)$ (amortized $O(1)$)	$O(n)$
insert	* $O(n)$	$O(n)$
delete	$O(n)$	$O(n)$

Figure 1: Items with * are those that may fail for a fixed-length implementation of an array, or those that trigger extension in extensible arrays

4.2 Stacks

- **Stack:** a **Last In First Out (LIFO)** buffer. Items get “stacked”, so we insert elements at the start, and we remove elements from the start



4.2.1 Actions on Stacks

- *empty*: check if stack is empty
- *push(x)*: insert an element x in the stack
- *peek*: check on the first element of the stack (the one that would be removed)
- *pop*: get the first element of the stack, removing it

4.2.2 Stack Implementation: Array vs Linked List

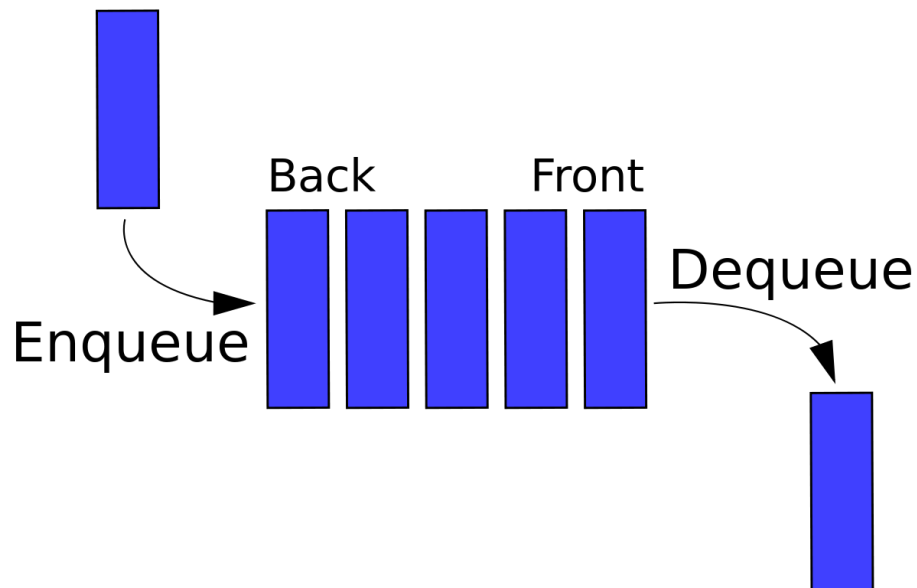
- Geeks for Geeks Implementation
- **Stacks as Arrays:** build stack by pushing and popping elements from the end of the array
 - ✓ easy to implement, no pointers required
 - × need extensible arrays, *push* can be $\mathcal{O}(n)$
- **Stacks as Linked Lists:** just add a node at the start to *push*, and remove it when *popping*
 - ✓ dynamic, easy to *push/pop*

× memory requirements from pointers

Operation	Extensible array impl	Linked list impl
empty	$O(1)$	$O(1)$
push	$* O(n)$ (amortized $O(1)$)	$O(1)$
peek	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$

4.3 Queues

- **Stack:** a **First in First Out (FIFO)** buffer. Items get added at the end, and removed from the start

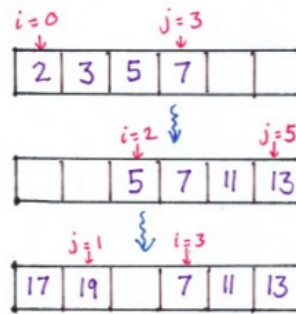


4.3.1 Actions on Queues

- *empty*: check if stack is empty
- *enqueue(x)*: insert an element x at the end of the queue
- *peek*: check on the first element of the stack (the one that would be removed)
- *dequeue*: get the first element of the queue, removing it

4.3.2 Queues as Wraparound Arrays

We keep track of the indices of the start (i) and the end (j) of the queue. If we add items, we increase j (modulo the length of the array); if we remove items, we increase i (modulo the length of the array). An empty queue will be given when $i = j$. A full queue will be given when $j = i - 1$. See this thorough example.



The initial queue is $[2, 3, 5, 7]$. If we *dequeue* twice, we remove the first 2 elements, and advance the front index, from $i = 0$ to $i = 2$. The resulting queue is $[5, 7]$. Then, apply *enqueue*(11) and *enqueue*(13) increases the end index, from $j = 3$ to $j = 5$, resulting in the queue $[5, 7, 11, 13]$. Finally, we *dequeue* (so $i = 3$) and then *enqueue*(17), *enqueue*(19) moves the end index further, but we wrap around, such that we go from $j = 5$ to $j = (5 + 2) \bmod 6 = 1$. The resulting queue thus becomes $[7, 11, 13, 17, 19]$.

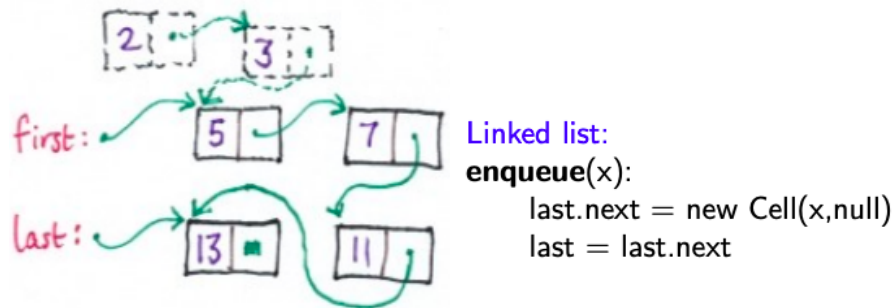
```

enqueue(x):
     $j = (j+1) \bmod |A|$ 
    if  $j = i$ 
        fail (or expand)
    else  $A[j] = x$ 
  
```

Figure 2: Enqueue in wraparound array queue implementation

4.3.3 Queues as Linked Lists

We have a linked lists with items in the queue. Keep pointers to the first and the last item of the list. Call these start and end pointer respectively. To dequeue, go to the node referenced by the start pointer, follow its reference to the second item in the queue, and make the start pointer reference this second item. Similarly, to enqueue, you create a new node that points nowhere, and make the last node of the queue point to this new node. Then, change the end pointer to point to the new node.



(a) Nodes 2 and 3 are nodes that have been dequeued from the LL

4.3.4 Comparison of Queue Implementations

Operation	Wraparound array impl	Linked-list impl
enqueue	* $O(n)$ (amortized $O(1)$)	$O(1)$
peek	$O(1)$	$O(1)$
dequeue	$O(1)$	$O(1)$

4.4 Sets and Dictionaries

- **Sets:** finite, unordered collection of objects containing elements of the same type. Any set should have the following actions:
 - *contains(x)*: check if x is in the set
 - *insert(x)*: insert x in the set
 - *delete(x)*: remove x from the set
 - *isEmpty*: check if set has elements
- **Dictionary:** maps keys to values. Any dictionary should have the following actions:
 - *lookup(x)*: given key x , returns corresponding value
 - *insert(x,y)*: insert key-value pair x,y in the dictionary
 - *delete(x)*: remove key (and corresponding value) x from dictionary
 - *isEmpty*: check if dictionary has elements
- **Aim of Sets and Dictionaries:** should provide a fast way of checking for the existence of a key (either *contains* or *lookup*)
- **Naively Implementing Sets and Dictionaries:** naively, can use a list to implement sets and dictionaries:

`names = ["James", "Mary", "Alex"]`

$birthyear = [(\text{"James"}, 1969), (\text{"Mary"}, 1420), (\text{"Alex"}, 3141)]$

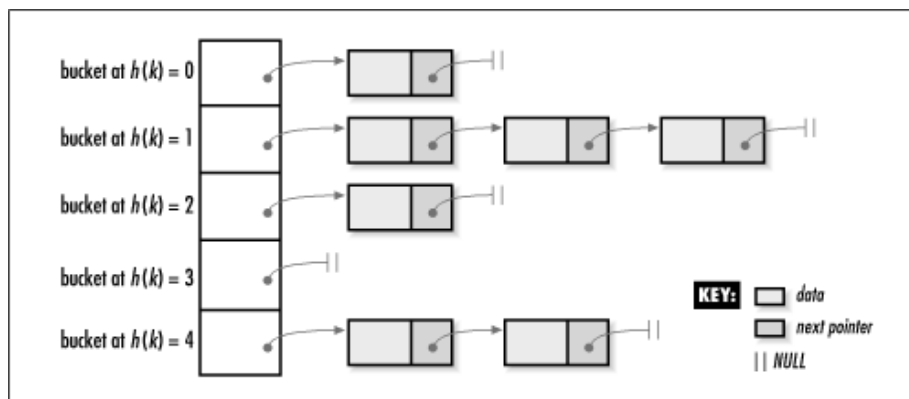
Looking up values would be $\Theta(n)$. Even if we sort and use **BinarySearch**, lookup time would be $\Theta(\lg(n))$. *insert/delete* will still take time

4.5 Hashing

- **Hash Tables:** given m key-value pairs, applies a *hash function* to the key, converting it into an integer (*hash code*) between 0 and $m - 1$ (so key s gets turned into $\#(s)$). Then, can store the value associated with s at index $\#(s)$ of an array of size m .
- **Hash Functions:** ideally want all hash codes to be equally likely.
 - × $\#(s) = s \bmod 2^k$: powers of two mean you will only consider last 2 characters of s
 - × $\#(s) = s \bmod 127$: numbers close to powers of 2 hash anagrams to the same hash code
 - ✓ $\#(s) = s \bmod 97$: primes far away from powers of 2 are good choices
- **Clash in Hash Tables:** it is possible that, if s, t are distinct keys, $\#(s) = \#(t)$. These clashes are quite likely, so we need to deal with them.
 - can use **perfect hashing** if the keys don't change, which removes any possibility of collision

4.5.1 Dealing with Clashes: Hash Buckets

- **Hash Buckets:** we want to use m hash codes. We have n key-value pairs. Create an array of size m . All values that hash into the same index of the array are kept in a form of linked list, with each item referencing the next. Then, the index of the array just holds a reference to the first of these items. Hence, at each index, we store a “bucket” of values.
 - if we have a set, we can just ignore the values



- **Load on Hash Table:** a number given by:

$$\alpha = \frac{n}{m}$$

- we can maintain a stable load if n gets too large by simply increasing the number of hash codes, and rehashing everything (good amortized cost)
- the average number of key comparisons for an unsuccessful lookup is precisely α , so $\Theta(\alpha)$ time for unsuccessful lookup (can be shown we get the same time for succesful lookup)

4.5.2 Dealing with Clashes: Open Addressing and Probing

- **Open Addressing:** store all items within the Hash Table (looking online, unaware if this means that all key value pairs are stored within the array, or just the values; from the lookup procedure it might seem that key value pairs are stored)
- **Probing:** resolves clashes when we use open addressing. Redefine hash function to $\#(k, i)$, $i \in [0, m - 1]$, and choose as the hash code of k the first hash such that said index in the array hasn't yet been occupied.
 - for *inserting*, if we have (k, v) , and $\#(k, 0)$, $\#(k, 1)$ already contain a value, but $\#(k, 2)$ is empty, then store v at index $\#(k, 2)$
 - for *lookup*, go through all possible hashing of k ($\#(k, 0)$, $\#(k, 1)$, etc ...) until an item is found which has key k
- **Analysis of Probing:**
 - ✓ low expected number of probes for *insert/lookup* ($\frac{1}{1-\alpha}$ for unsuccessful, less if succesful)
 - ✓ no need for pointers, can invest memory in additional hash codes (faster lookup for same memory compared with buckets)
 - × *deleting* is hard
 - × designing probing function not straightforward

5 Week 5 - Balanced Trees

5.1 Motivation

- **Sets and Dictionaries Action Runtime:** *lookup/insert/delete* is $\Theta(1)$ on the average, but $\Theta(n)$ **worst case**
- **List Action Runtime:** many worst case actions run in $\Theta(1)$, but *insert/delete* is $\Theta(n)$ average case
- **Balanced Trees:** data structures which allow us to obtain an average case of $\Theta(\lg(n))$ **for all operations**

5.2 Binary Trees

- **Binary Tree:** tree structure, such that:
 - every node x has a *left* ($L(x)$) and *right* ($R(x)$) subtree
 - each node is labelled with a *key*
 - * if we implement a dictionary, each node contains a key and its value
 - for any node x :
 - * any child from its left subtree will have a smaller key than x

$$\forall y \in L(x), y.key < x.key$$

- * any child from its right subtree will have a larger key than x

$$\forall z \in R(x), z.key > x.key$$

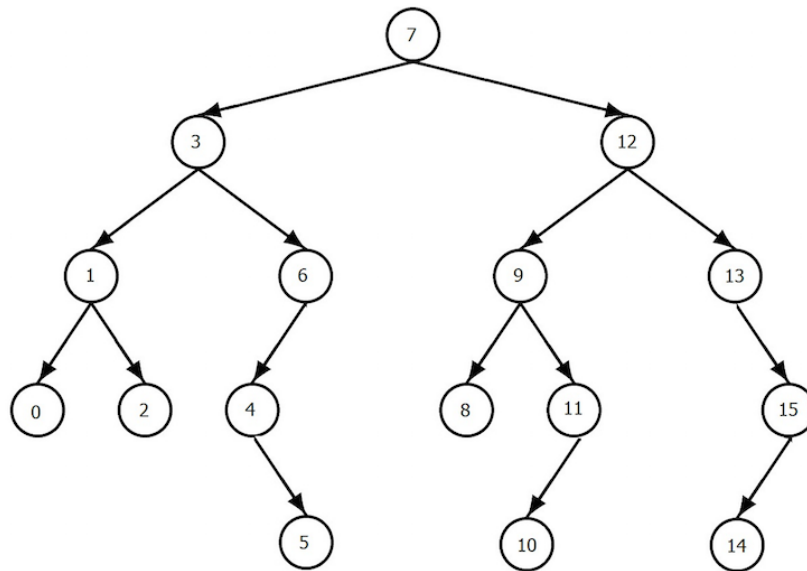
```

contains'(x,k):
  if x = null then return False
  else if x.key = k then return True
  else if k < x.key then return contains'(x.left,k)
  else return contains'(x.right,k)

contains(k):
  return contains'(root,k)

```

Figure 4: Contains algorithm in a binary tree. Employs the structure of left and right branches, alongside recursion to find a key within the tree



- **Node Depth:** number of edges from the root to the node
- **Node Height:** number of edges from the node to the deepest leaf
- **Tree Depth:** number of edges from root node to the deepest leaf

5.2.1 Contains

- **Perfectly Balanced Tree:** a tree of n nodes and depth d is *perfectly balanced* only if $n = 2^d - 1$ (all non-leaf nodes have 2 children, and all leaf nodes are at depth d)

- **Contains Analysis:** depending on the “type” of tree, *contains* has different performance:
 - **Perfectly Balanced Tree:** by definition $d = \lg(n + 1)$, so at most *contains* will have to traverse the whole tree, so it is $\mathcal{O}(\lg(n))$
 - **Not-Too-Unbalanced Tree:** trees with max depth $\leq 2\lceil \lg(n) \rceil$, *contains* is also $\mathcal{O}(\lg(n))$

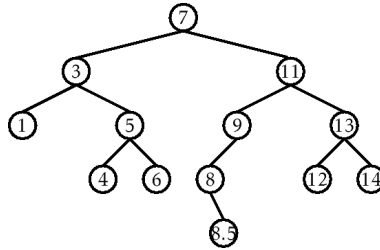


Figure 5: Not-Too-Unbalanced Tree. We have $d_{max} = 5$, and $n = 13$, so $d_{max} = 5 \leq 8 = 2 \times \lceil \lg(13) \rceil$

- **Worst Case:** any tree in which all nodes are organised in a single file (i.e every node is a left child of its parent). Then *contains* will be $\Theta(n)$ in this worst case.

5.2.2 Insert

```

insert'(x,k):
  if x.key = k then return KeyAlreadyPresent
  else if k < x.key then
    if x.left = null then x.left = new Node(k)
    else insert'(x.left,k)
  else
    if x.right = null then x.right = new Node(k)
    else insert'(x.right,k)

insert(k):
  if root = null then root = new Node(k)
  else return insert'(root,k)
  
```

Figure 6: To insert a new key, just need recursively call insert on children nodes, until we find a suitable place for the key which allows it to satisfy the binary tree rules (right tree if bigger, left subtree if smaller)

- **Balanced/Not-Too-Unbalanced Tree:** $\mathcal{O}(\lg(n))$ by similar arguments as before

- **Worst Case:** $\Theta(n)$

5.2.3 Delete

Let y be a node, and $y.key = j$. If we want to delete j , we consider 3 cases:

1. **y is a leaf**
 - remove y
2. **y has 1 child**
 - *elide* y (delete y by making the parent of y the parent of y 's child)

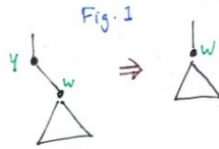


Figure 7: In removing y , we move its child w to occupy its place

3. **y has 2 children**
 - let z be the leftmost node of $R(y)$
 - make $y.key = j = z.key$
 - apply *delete* to z

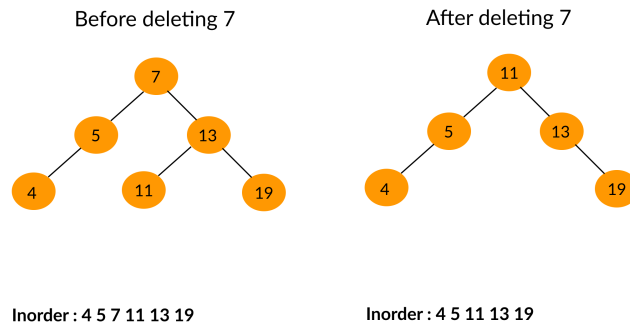


Figure 8: Here $j = 7$, and $z.key = 11$

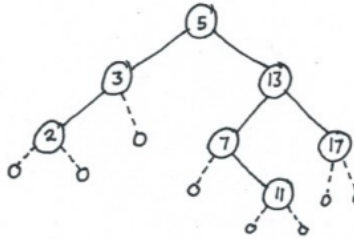
Again, $\mathcal{O}(\lg(n))$ if tree is slightly unbalanced, but worst case $\Theta(n)$

5.2.4 Binary Trees Summary

If the binary tree is only slightly unbalanced, then we can achieve an average case $\mathcal{O}(\lg(n))$. The issue occurs when the tree becomes unbalanced as a consequence of inserting and deleting.

5.3 Red-Black Trees

- **Red-Black Trees:** a special type of tree, which uses a set of rules to guarantee that, even after *insert* and *delete*, it remains relatively balanced. Rebalancing can be done in $\mathcal{O}(\lg(n))$, so even worst case actions will be $\mathcal{O}(\lg(n))$
- **Trivial Nodes:** instead of having plain leaf nodes, red-black trees have *trivial nodes* (leaf nodes point to trivial nodes, instead of just pointing to *null*)
 - remaining nodes are *proper nodes*
 - resulting tree is an *extended tree*



- **Red-Black Tree Rules:** a red-black tree is an extended tree composed of red and black nodes. Must satisfy the following rules:
 1. root and trivial nodes are black
 2. same number of black nodes along any path from root to leaf
 3. no 2 consecutive red nodes along any path from root to leaf
- **Depth of Red-Black Trees:** if the number of black nodes along any path is b , then:
 - $d_{min} = b$ (only black nodes)
 - $d_{max} = 2b - 1$ (alternating black and red nodes)
- **Balance of Red-Black Trees:** it can be shown that $b \leq \lg(n + 1) + 1$. Since $d_{max} = 2b - 1$, it follows that any path length is at most of length:

$$2\lg(n + 1) + 1$$

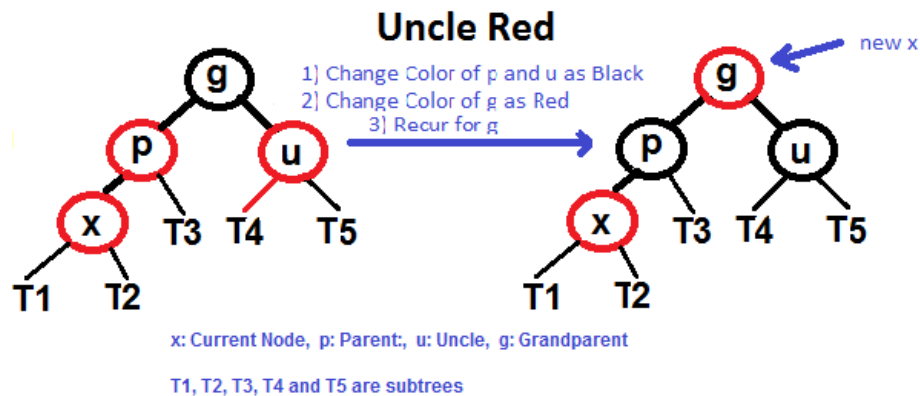
This means that we can guarantee $\mathcal{O}(\lg(n))$ operations **given that the rules are kept**

- **Contains in Red-Black Tree:** worst case time is guaranteed $\Theta(\lg(n))$
- **Delete in Red-Black Tree:** we delete as normal. Complications arise if the node to be deleted is black, as then a path will be one black short.

- if the node that occupies the deleted node's position is red, just turn it black
- if deleting procedure reaches the root, delete root
- more complex, albeit fixable scenarios not covered

5.3.1 Insert and the Red-Uncle Rule

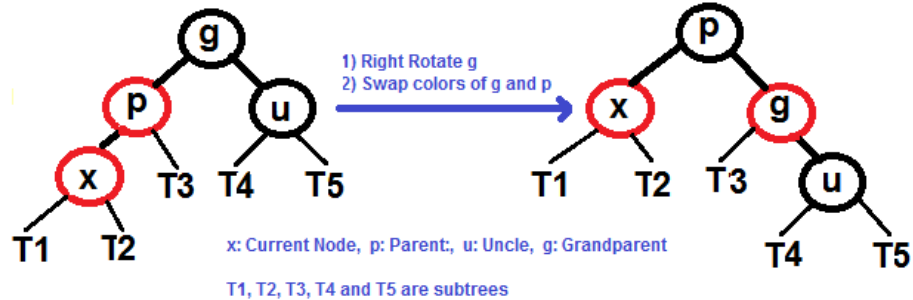
- **Inserting in Red-Black Tree:** we insert a node as with a binary tree. Inserted node must be *red*, and have 2 trivial leaves. 2 problems:
 - new node can have a red parent
 - new node might be the root
- **Red-Uncle Rule:** used to guarantee that insertion preserves the Red-Black Tree rules. If we insert a new node x , and both its parent and uncle are red nodes, then make them black. Then, make the grandparent of x red. We repeat this procedure until no longer applicable.



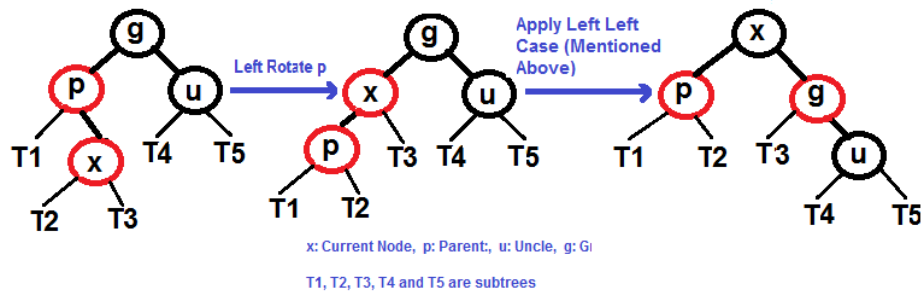
- **Steps After Red-Uncle Rule:** after applying the red-uncle rule as much as possible, we can be in 3 situations:
 1. **Legal Tree Reached**
 2. **Root Becomes Red**
 - turn it black (preserves rules, as all paths increase number of black nodes by 1)
 3. **No Red Uncle**
 - if we inserted x , then the grandparent of x must have **4 nearest black descendants**. Nodes can be shuffled, preserving rules. See here.

- shuffling can be done in $\mathcal{O}(1)$ time

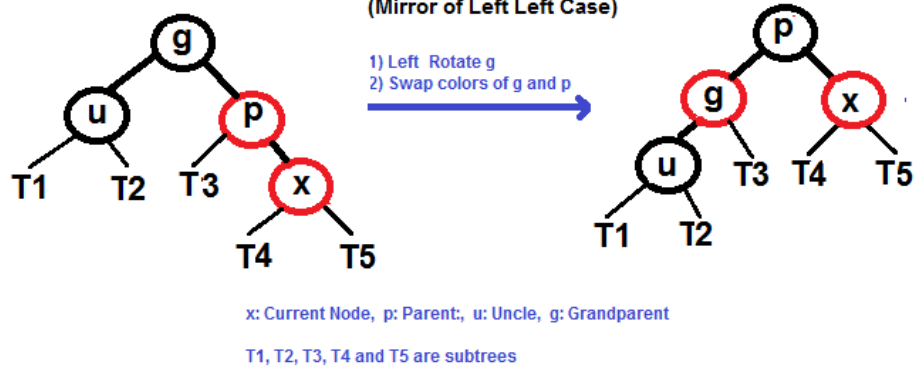
Uncle Black and Left Left Case



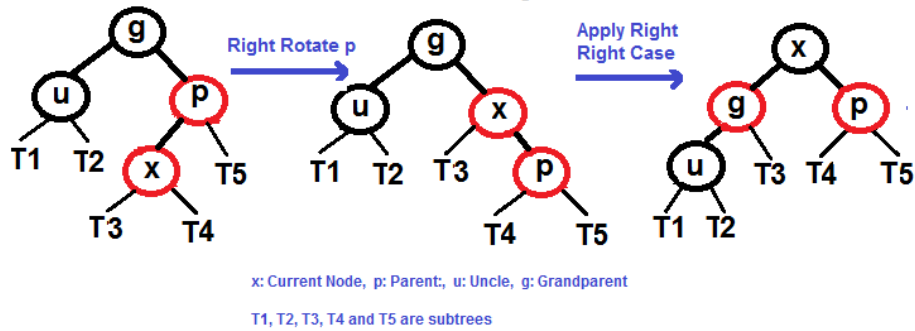
Uncle Black and Left Right Case



Uncle Black and Right Right Case (Mirror of Left Left Case)



Uncle Black and Right Left Case (Mirror of Left Right Case)



5.4 Summary of Balanced Trees

- **Benefit of Balanced Trees:** allow us to implement data structures with worst-case operation runtime $\mathcal{O}(\lg(n))$
- **Alternatives to Red-Black Trees:** can also use AVL trees
 - AVL more balanced (faster lookup)
 - Red-Black Trees have faster *insert/delete*

5.5 Data Structures Review

- **Abstract Data Structures:** lists, stacks, queues, sets, dictionaries
- **Concrete Implementations of Data Structures:** Extensible Arrays, Linked Lists, Hash Tables, Red-Black Trees

6 Week 5 - Divide-Conquer-Combine and the Master Theorem

6.1 Runtime of Recursive Algorithms

- **Seen Recursive Algorithms:**
 - **MergeSort:** sorts an array by splitting it, sorting the halves, and then merging the sorted halves
 - **insert:** in binary trees, you apply **insert** in the subtree of a node until an appropriate position is found
- **Divide, Conquer and Combine:** recursive algorithms can be broken down into 3 parts:
 - **Divide** problems into subproblems
 - * compute size of subarrays
 - **Conquer** (aka solve) the subproblems
 - * recursively sort each subarray
 - **Combine** results, to solve the main problem
 - * apply *merge* to combine the subarrays
- **Recursive Runtime:** since in recursion we apply the algorithm for smaller subproblems, the runtime of such algorithms will be a **recursive formula**.
 - for **MergeSort**, let $T(n)$ denote a worst case runtime. Then the runtime will be:

$$T(n) = \begin{cases} C & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + F(n) & n \neq 1 \end{cases}$$

- C is just a constant (denoting time taken to **MergeSort** a unit list)
- $F(n)$ is the time taken to **Merge** lists of combine length n
- **Simplifying Recursive Runtime:** we don't care about the exact value of $T(n)$: we only want to know the order of growth. Thus, we can use asymptotics to simplify the expression (can remove floors/ceilings, as asymptotically don't matter, can remove unwanted functions, etc...):

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n \neq 1 \end{cases}$$

- at this point, we can use techniques like mathematical induction to solve for $T(n)$ (see slides¹)

¹Week 5, Lecture 10, Slide 8

6.2 The Master Theorem

- **The Master Theorem:** allows us to “solve” for runtime, given a recursive runtime. If the runtime is of the form:

$$T(n) = \begin{cases} \Theta(1) & n \leq n_0 \\ aT\left(\frac{n}{b}\right) + \Theta(n^k) & n > n_0 \end{cases}$$

then, by the Master Theorem, and letting $e = \log_b(a)$,

$$T(n) = \begin{cases} \Theta(n^e) & e > k \\ \Theta(n^k \times \lg(n)) & e = k \\ \Theta(n^k) & e < k \end{cases}$$

- **Intuition Behind Master Theorem:** comparing the constants with our recursion for MergeSort:

- a gives the number of subproblems
- b gives the size of each subproblem
- n^k time required to solve subproblem
- **$e = k$:** basically means $a = b^k$
 - * increasing a increases the work required as we descend the tree (more subproblems to solve)
 - * increasing b decreases the size of the subproblem by factor b , so work required to divide/combine will decrease by b^k
 - * if $a = b^k$, the amount of work balances out, so it will be similar across all levels. There are $\lg(n)$ levels, and the total work to be done at the top level if $T(n) = n^k$, so, if $a = b^k$:

$$T(n) = n^k \times \lg(n)$$

- **$e > k$:** basically means $a > b^k$
 - * the number of subproblems increases faster than the rate at which subproblems decrease in size
 - * as we go down the tree, we expect more work, so the total cost is dominated by the cost of solving base cases
 - * at the top level, merging all the branches is $\Theta(n^k)$ work
 - * there are a total of about $\log_b(n)$ levels (b is size of subproblem, so $n \approx b^d$ for a tree of depth d)
 - * let $r = \frac{a}{b^k}$. At the bottom level, the proportion of bottom level work to top level work is:

$$r^{\log_b(n)} = b^{\log_b(r) \log_b(n)} = b^{\log_b(n) \log_b(\frac{a}{b^k})} = n^{\log_b(a) - \log_b(b^k)} = n^{e-k}$$

* thus, the total work done at the bottom level is:

$$\Theta(n^k) \times \Theta(n^{e-k}) = \Theta(n^e)$$

– $e < k$: basically means $a < b^k$

- * the work in merging subproblems is greater than the work for solving the subproblems
- * thus, most of the work will be done when merging all these big subproblems at the top, rather than in solving them at the bottom
- * at the top we do $\Theta(n^k)$ work
- * if $r = \frac{a}{b^k}$, then work decreases by factor r as we go down the tree, with proportion of work at the bottom to work at the top being roughly:

$$1 + r + r^2 + \dots \leq \frac{1}{1-r} \in \mathbb{R}$$

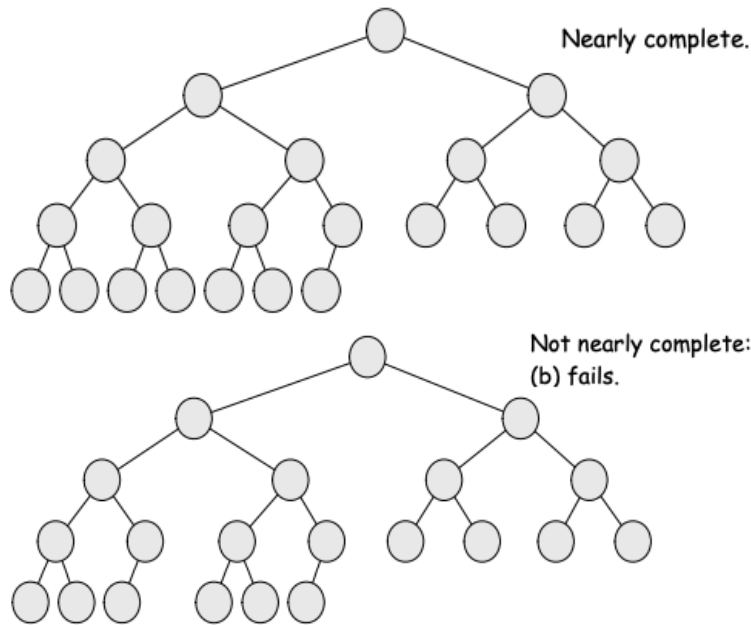
* so total work in the end will be roughly:

$$\frac{1}{1-r} \times \Theta(n^k) = \Theta(n^k)$$

7 Week 7 - The Heap Data Structure

7.1 The Heap

- **Nearly Complete Binary Tree:** a binary tree of depth h , such that:
 - $\forall d \in [0, h - 1]$, every level has 2^d nodes (so every level is completely filled)
 - all leaf nodes ($d = h$) are as far left as possible



- **The Heap Data Structure:** a nearly complete binary tree, where every parent node is greater than or equal to each of its child nodes
 - can be easily stored in an array
 - no total ordering of items in array
 - ideal for quickly finding maximum value of an array
- **Heaps as Arrays:** a heap with n elements can be stored by reading indices left-to-right from the heap. Mathematically, the index in the array of the j^{th} item in the i^{th} row is:

$$index = (2^i - 1) + j - 1$$

- $2^i - 1$ are all the items in the level above
- there are $j - 1$ items before item j

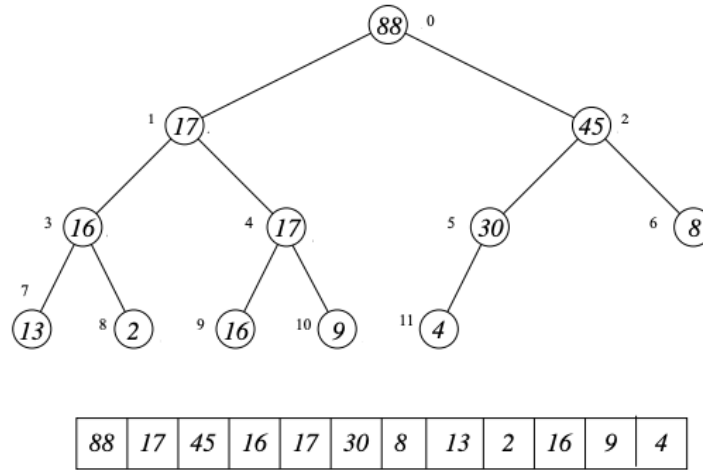


Figure 9: We can label nodes by reading left to right in the order that they appear.

- **Height of Heap:** all leaf nodes are at height $h - 1$ or h (root node is $h = 0$). Thus, any heap of n nodes must satisfy:

$$2^h \leq n < 2^{h+1} - 1 \implies h \leq \lg(n) < h + 1$$

But then it follows that:

$$\lg(n) - 1 < h \leq \lg(n)$$

Any operation that depends on the height of the heap will probably have operations with runtime $\Theta(\lg(n))$

7.2 Operations on Heaps

7.2.1 Heap-Maximum

Returns the maximum value of a heap, which can be easily done in $\Theta(1)$ time

7.2.2 Max-Heapify

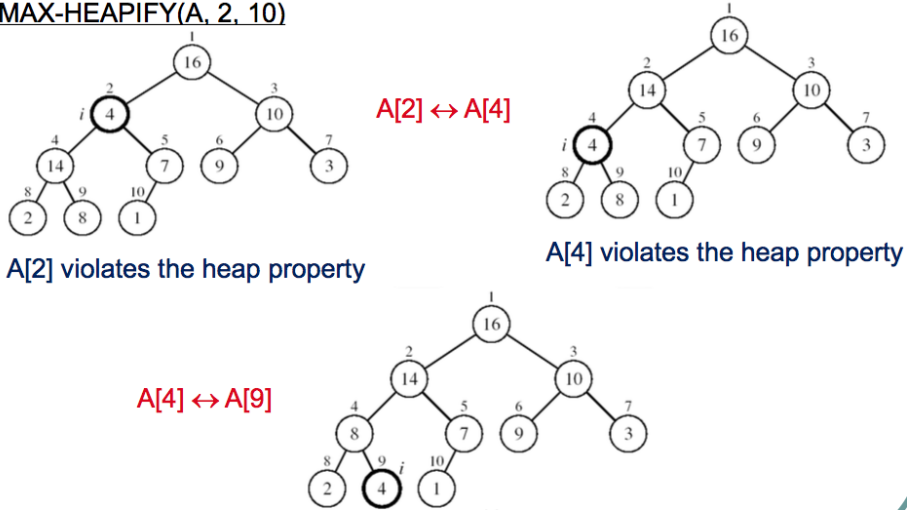
- **Max-Heapify:** if a node at index i violates the heap property, use Max-Heapify to restore the heap property
 - in essence, compares i with its 2 children. Select the largest of the children, call it x , and then swap i and x . Now, i is a child of x . Call Max-Heapify on i .

Algorithm Max-Heapify(A, i)

1. $\ell \leftarrow \text{Left}(i)$
2. $r \leftarrow \text{Right}(i)$
3. $\text{largest} \leftarrow i$
4. **if** $\ell < A.\text{heap_size}$ **and** $A[\ell] > A[i]$
5. $\text{largest} \leftarrow \ell$
6. **if** $r < A.\text{heap_size}$ **and** $A[r] > A[\text{largest}]$
7. $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. exchange $A[i]$ with $A[\text{largest}]$
10. Max-Heapify($A, \text{largest}$)

Figure 10: $L(i)$ and $R(i)$ respectively correspond to the index of the left and right child of i . largest contains the index of the biggest key out of i , $L(i)$ and $R(i)$

MAX-HEAPIFY($A, 2, 10$)



- **Max-Heapify Runtime Analysis:** intuitively, we expect that in the worst case, we have to traverse all of the height of the tree, at each step doing 2 comparison and 1 swap. Thus, we execute $\Theta(1)$ operations at most/least h times, so we expect runtime to be:

$$h \times \Theta(1) = \Theta(h) = \Theta(\lg(n))$$

- more formally, we have a recursive formula:

$$T_{Max-Heapify}(h) \leq \begin{cases} T_{Max-Heapify}(h-1) + \mathcal{O}(1) & h \geq 1 \\ \mathcal{O}(1) & h = 0 \end{cases}$$

(formula is identical for Ω)

- it then follows that, since $h \leq \lg(n)$:

$$T_{Max-Heapify}(h) \leq (h+1)\mathcal{O}(1) \implies T_{Max-Heapify}(h) \in \mathcal{O}(h) = \mathcal{O}(\lg(n))$$

- similarly, since $h \geq \lg(n) - 1$, and we do $\Omega(1)$ work at each step for h comparisons:

$$T_{Max-Heapify}(h) = h\Omega(1) \implies T_{Max-Heapify}(h) \in \Omega(h) = \Omega(\lg(n))$$

- thus, the runtime of **Max-Heapify** is:

$$\Theta(\lg(n))$$

7.2.3 Heap-Extract-Max

- **Heap-Extract-Max**: like *pop*, returns (and in the process removes) the largest value of the heap
 - we can get the max element in $\Theta(1)$ time
 - swap the max element ($A[0]$), with the last element of the heap ($A[A.length - 1]$)
 - remove the last element of the heap ($A.length - = 1$)
 - call **Max-Heapify**(0) (so we heapify at the root, to ensure that the resulting object maintains the Heap property)
 - most of the work is done by **Max-Heapify**, so the runtime for **Heap-Extract-Max** is:

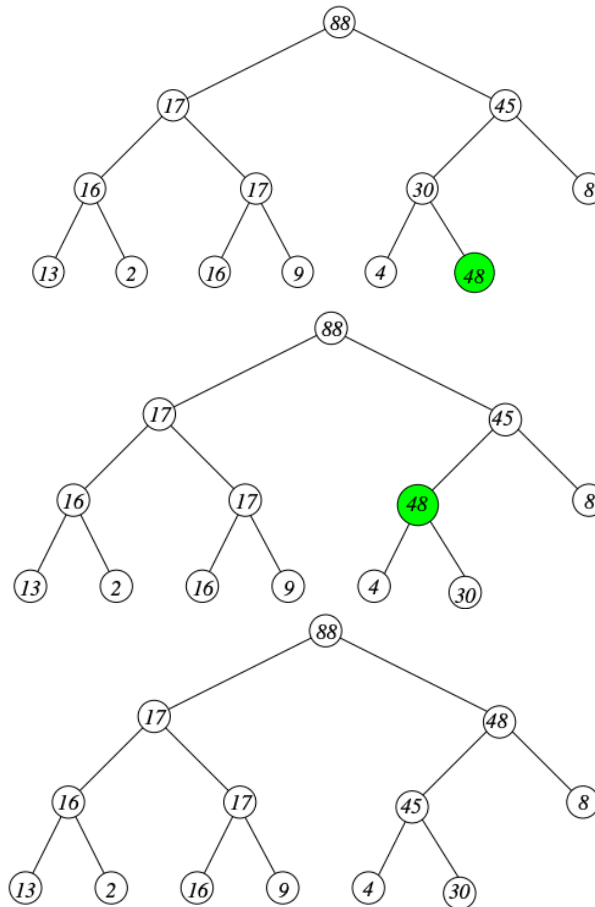
$$\Theta(\lg(n))$$

7.2.4 Max-Heap-Insert

- **Max-Heap-Insert**: insert new item within heap (preserving heap property)
 - in essence, inserts element at the end of the heap. Then, “bubble” the item up, by swapping it with its parent if the parent is smaller

Algorithm Max-Heap-Insert(A, k)

1. $A.heap_size \leftarrow A.heap_size + 1$
2. $A[heap_size - 1] \leftarrow k$.
3. $j \leftarrow heap_size - 1$
4. **while** ($j \neq 0$ and $A[j] > A[Parent(j)]$) **do**
5. exchange $A[j]$ and $A[Parent(j)]$
6. $j \leftarrow Parent(j)$



- **Max-Heap-Insert Runtime Analysis:** it is easy to see that, in a worst case, we insert an item larger than every other element of the heap. Then we need to compare and swap this item with all of its parents, so the runtime for Max-Heap-Insert is:

$$\Theta(\lg(n))$$

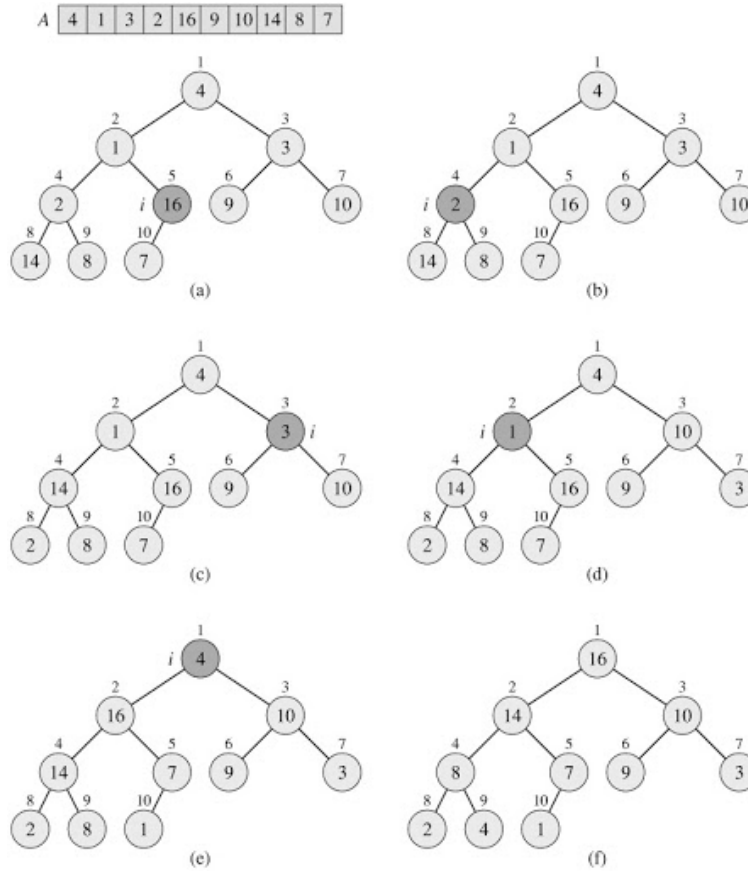
7.2.5 Build-Max-Heap

- **Build-Max-Heap**: builds a heap from an unsorted array
 - in essence, we want to go over every non-leaf node, from bottom to the top, applying **Max-Heapify**. This will ensure that we preserve the Heap property from bottom up.
 - good video on Build-Max-Heap

Algorithm Build-Max-Heap(A)

1. $A.heap_size \leftarrow A.length$
2. **for** $i \leftarrow \lfloor A.heap_size/2 \rfloor - 1$ **downto** 0
3. **Max-Heapify**(A, i)

Figure 11: Since about half of the nodes will be leaf nodes, we only need to iterate over $\lfloor A.heap_size/2 \rfloor$ items



- **Build-Max-Heap Runtime Analysis:** notice that we call **Max-Heapify** a different number of times depending on the height: 1 time when at the root (height h), 2 times when height is $h - 1$, 4 times when height is $h - 2$. In general, at height $l \in [1, h]$, we call **Max-Heapify**:

$$\left\lceil \frac{n}{2^{1+l}} \right\rceil$$

times. Moreover, we know that, at height l , **Max-Heapify** has a runtime of $\mathcal{O}(l)$. Thus, the total runtime is:

$$\sum_{l=1}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{n}{2^{1+l}} \right\rceil \times \mathcal{O}(l) = \mathcal{O}(n)$$

(we have used identity A.8 from CLRS at the end)

7.2.6 Summary of Heap Operations

Parent (i)	$\mathcal{O}(1)$
Left (i)	$\mathcal{O}(1)$
Right (i)	$\mathcal{O}(1)$
Heap-Maximum (A)	$\mathcal{O}(1)$
Max-Heapify (A, i)	$\Theta(\lg(n))$
Heap-Extract-Max (A)	$\Theta(\lg(n))$
Max-Heap-Insert (A, k)	$\Theta(\lg(n))$
Heap-Increase-Key (A, i, k)	$\Theta(\lg(n))$
Build-Heap (A)	$\Theta(n)$

Also, go [here](#) for all the pseudocode.

Lastly, **Heap-Increase-Key**(A, i, k) basically changes the key stored at $A[i]$ by k (assuming $k > A[i]$). Very similar to **Max-Heap-Insert** in terms of functioning.

7.3 HeapSort

- **HeapSort:** allows us to extract a total ordering of the elements in the heap
 - we extract the largest item of the heap, and swap it with the last element of the array. Then, decrement the size of the heap (so now it contains every element except the original largest element). Apply **Max-Heapify** (if we use **Heap-Extract-Max**, in removing the max

element we preserve the heap property already). Repeat until size of the heap is 0.

- this (Brilliant) and this (HappyCoders) are great articles on HeapSort (and the Heap in general)

Algorithm HeapSort(A)

1. $n \leftarrow A.length$
2. Build-Max-Heap(A)
3. **for** $i \leftarrow n - 1$ **downto** 0
4. $v \leftarrow \text{Heap-Extract-Max}(A)$
5. $A[A.heap_size] \leftarrow v$

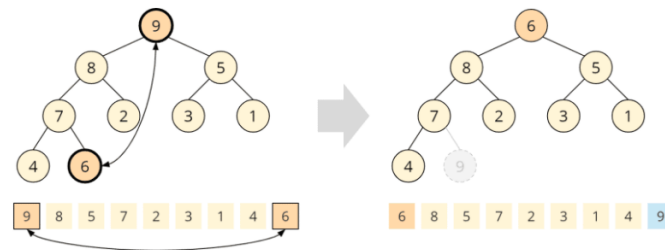


Figure 12: We swap the max and the last item in the array, and stop considering the max element as part of the heap

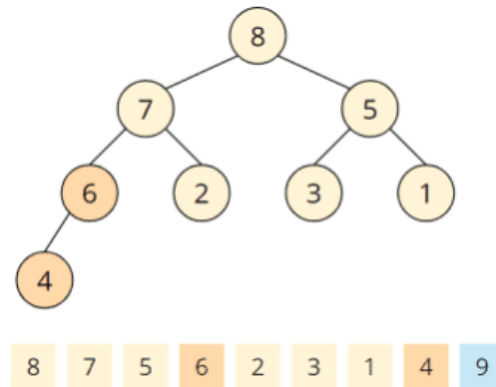


Figure 13: After, apply Max-Heapify to obtain a legal max heap

- **Heapsort Runtime Analysis:** most of the work comes from using Heap-Extract-Max, which for a heap of m nodes has runtime $\mathcal{O}(\lg(m))$.

Since we apply it a total of n times, the runtime is:

$$\begin{aligned} & \mathcal{O}\left(\sum_{m=1}^n \lg(m)\right) \\ &= \mathcal{O}\left(\lg\left(\prod_{m=1}^n m\right)\right) \\ &= \mathcal{O}(\lg(n!)) \end{aligned}$$

It is easy to show that $n^{\frac{n}{2}} \leq n! \leq n^n$, so taking logs, this allows us to justify that **HeapSort** is $\mathcal{O}(n \lg(n))$ and $\Omega(n \lg(n))$ (achieved if input array is reversed) algorithm

- created (alongside Heap) by J.W.J Williams
- in place algorithm
- it is **unstable**: keys that appear in an order within the original array are not guaranteed to have this order preserved after **HeapSort** is applied

7.4 Priority Queues as Heaps

- **Priority Queue**: collection in which items have an associated key (larger key = more priority)
 - you remove those items with highest priority
 - used when resources need to be managed (i.e printing), and priorities should be given to users

8 Week 8 - Quicksort

8.1 Overview

- **Quicksort**: a divide and conquer algorithm for sorting, developed by Tony Hoare
 - the base case is an array of length 2, for which nothing is done. Otherwise, select a pivot p , with which the array is partitioned into 2 subarrays: one containing elements less than p , and the other containing elements larger than p . Then, apply **QuickSort** on these 2 subarrays.
 - an integral part of the algorithm is the **Partition** step. We let the pivot be the last element of the array (say at index r), and we keep track of 2 constants, i and j . At each step, we will be increasing j by 1, until it eventually reaches $j = r - 1$. If we find that $A[j]$ is less than or equal to the pivot, we then increase i by 1, and swap $A[i]$ and $A[j]$. If $i = j$, this changes nothing. However, if $i < j$, then this means that when j advanced, i didn't move, meaning that j encountered an element greater than the pivot. Hence, by swapping, we ensure that elements smaller than the pivot stay to the left, and elements greater than the pivot stay to the right. Once $j = r - 1$, we can just place the pivot at index $i + 1$ (by exchanging $A[i + 1]$ and $A[r]$). This means that we will have all elements to the left of the pivot being less than or equal to it, and all of the elements to its right will be greater than it. **Partition** will then return the index $i + 1$, so that we know where to divide the arrays for the next iteration of **QuickSort**.
 - it is important to note that in all sources I found online, the **Partition** is a bit different, with the pivot being anywhere in the array, and with i and j being *Left* and *Right* pointers that begin at either end of the array. If interested, see:
 - * python3, gives overview, has an interactive animation, and even has some testing. Contains Python implementation.
 - * a good article, works through an example, and even has a video. Contains JavaScript implementation.

Algorithm QuickSort(A, p, r)

1. **if** $p < r$ **then**
2. $split \leftarrow \text{Partition}(A, p, r)$
3. QuickSort($A, p, split - 1$)
4. QuickSort($A, split + 1, r$)

Figure 14: Here p represents the left index from which we will apply QuickSort. Indeed, the subarray that we will partition will be the part of the array that goes from index p to index r . If we want to QuickSort the whole array, $p = 0$ and $r = A.size$. $split$ is the index of the pivot resulting from applying partition; it tells us how to split A into 2 subarrays.

Algorithm Partition(A, p, r)

1. $pivot \leftarrow A[r].key$
2. $i \leftarrow p - 1$
3. **for** $j \leftarrow p$ **to** $r - 1$ **do**
4. **if** $A[j] \leq pivot$
5. $i \leftarrow i + 1$
6. exchange $A[i]$ and $A[j]$
7. exchange $A[i + 1]$ and $A[r]$
8. **return** $i + 1$

Figure 15: Check the slides (Lecture 13, Slide 7) to check for correctness of Partition

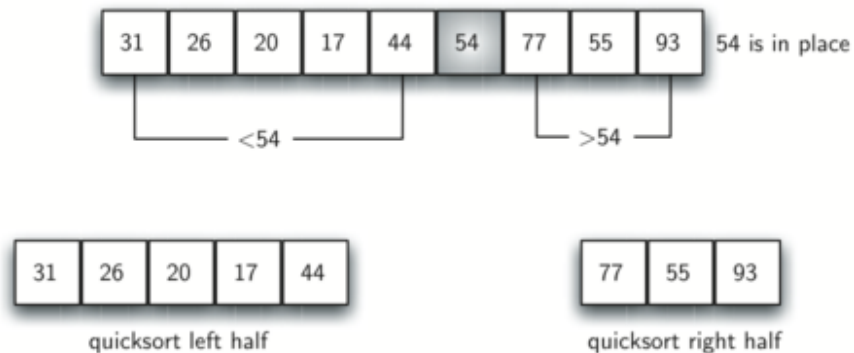



Figure 16: Array after using Partition with pivot 54. We would then apply quicksort to both of the arrays at either side of 54

Iteration 1 ($j = 0$)

2	9	19	8	20	5	16
---	---	----	---	----	---	----

i j

We begin with pivot 16,
and $j = p = 0$.



2	9	19	8	20	5	16
---	---	----	---	----	---	----

i j

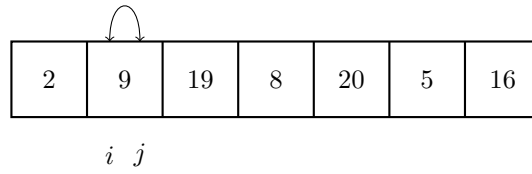
Since $A[j = 0] = 2 \leq 16$,
we increase the counter
of i ($i = 0$), and swap
 $A[i = 0]$ with $A[j = 0]$.

Iteration 2 ($j = 1$)

2	9	19	8	20	5	16
---	---	----	---	----	---	----

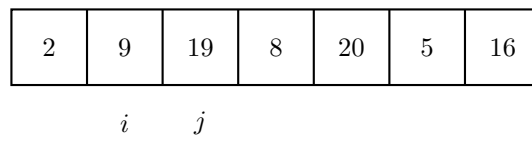
i j

Increase j ($j = 1$)

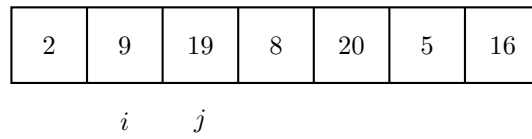


Since $A[j = 1] = 9 \leq 16$,
we increase the counter
of i ($i = 1$), and swap
 $A[i = 1]$ with $A[j = 1]$.

Iteration 3 ($j = 2$)



Increase j ($j = 2$)




Since $A[j = 2] = 19 > 16$,
we do nothing.

Iteration 4 ($j = 3$)

2	9	19	8	20	5	16
	i		j			

Increase j ($j = 3$)

2	9	19	8	20	5	16
	i		j			



Since $A[j = 3] = 8 \leq 16$,
we increase the counter
of i ($i = 2$), and swap
 $A[i = 2]$ with $A[j = 3]$.

Iteration 5 ($j = 4$)

2	9	8	19	20	5	16
	i		j			

Increase j ($j = 4$)

2	9	8	19	20	5	16
---	---	---	----	----	---	----

i j

Since $A[j = 4] = 20 > 16$,
we do nothing.

Iteration 6 ($j = 5$)

2	9	8	19	20	5	16
---	---	---	----	----	---	----

i j

Increase j ($j = 5$)

2	9	8	19	20	5	16
---	---	---	----	----	---	----

i j

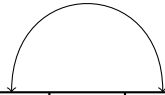
Since $A[j = 5] = 5 \leq 16$,
we increase the counter
of i ($i = 3$), and swap
 $A[i = 3]$ with $A[j = 5]$.

Final Swap

2	9	8	5	20	19	16
			i		j	

Since in the previous iteration we reached $j = 5$, the for loop terminates

2	9	8	5	20	19	16
			i		j	



We swap $A[i + 1 = 4]$ and $A[r = 6]$, and we have partitioned the array successfully!

Result

2	9	8	5	16	19	20
---	---	---	---	----	----	----

8.2 Quicksort Runtime Analysis

- **Runtime Intuition:**

1. at the top level, no matter what, we will do $\Theta(n)$ work (this is work in partitioning the whole array, analogous to merging in **MergeSort**)
2. how **Partition** splits the array can heavily influence the runtime:

- (a) if **Partition** places the pivot close to the centre of its subarray, then **QuickSort** behaves like **MergeSort** (constantly halving the array to sort), so we expect a runtime of $\Theta(n \lg(n))$
 - if the sides are balanced we get a runtime equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

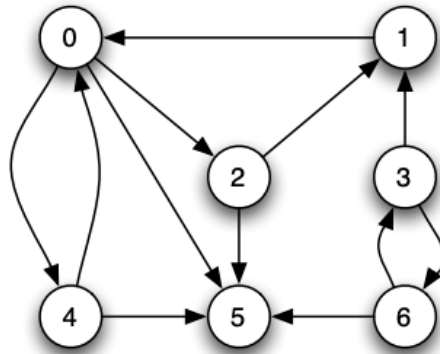
- (b) alternatively, **Partition** can be heavily unbalanced (for example, 90% of array is to the left of the pivot). In this case, we have to do a lot of work to reduce the size of the array (think that in each iteration of partition the subarray to sort becomes shorter by 1 or 2 elements)
 - (c) naturally, anything else can happen in between of these 2 “extreme” cases
- **QuickSort Runtime:** a more detailed analysis of runtime can be found here (or in the slides, Lecture 13, Slide 9). In summary:
 - **Worst Case Runtime:** $\Theta(n^2)$
 - **Average Case Runtime:** $\Theta(n \lg(n))$
 - **Best Case Runtime:** $\Theta(n \lg(n))$
 - **Consequences of Runtime on QuickSort:** **QuickSort** is very fast (has a smaller constant than **MergeSort**), but suffers in worst case scenarios (for example, if the array is already sorted, or nearly so)
 - **Improving QuickSort:**
 - pick random pivot (**RandomQuickSort**)
 - better partitioning
 - **InsertSort** for smaller/sorted arrays

9 Week 9 - Graphs: Representation and Searching

9.1 Graphs

- **Graph:** structure composed by a set of vertices V and a set of edges E
 - edges join vertices, so $E \subseteq V \times V$
 - a graph G can be described by $G = (V, E)$
 - we will use the convention that $n = |V|$ and $m = |E|$
- **Directed Graph:** edges go from one vertex to another one (i.e can go from v to w , but not the other way)
- **Undirected Graph:** edges, such that:

$$(v, w) \in E \iff (w, v) \in E$$



$$E = \{(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)\}. \quad V = \{0, 1, 2, 3, 4, 5, 6\}$$

Figure 17: Directed graph

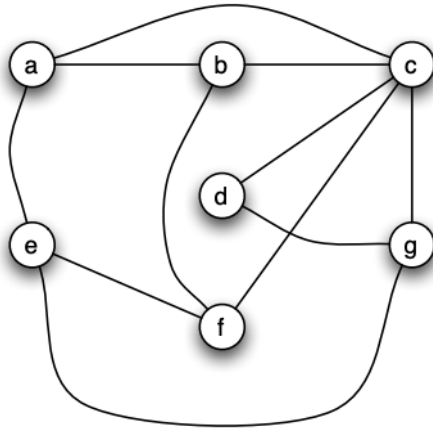


Figure 18: Undirected graph

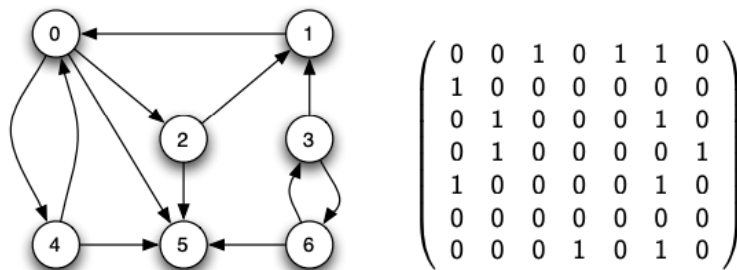
- **Degree of a Vertex:** if v is a vertex,
 - $in(v)$ denotes the *in-degree* of v (number of edges that end in v)
 - $out(v)$ denotes the *out-degree* of v (number of edges stemming from v)

9.2 Representing Graphs

9.2.1 Adjacency Matrix

Given n vertices, create an $n \times n$ matrix A , such that, for any 2 vertices i, j :

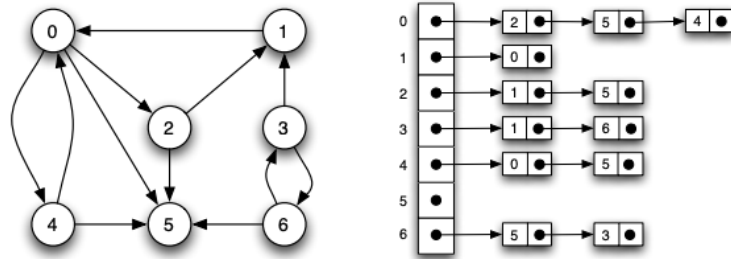
- if (i, j) is an edge, $A_{ij} = 1$
- $A_{ij} = 0$ otherwise



If G is an undirected graph, the adjacency matrix for G will be symmetric.

9.2.2 Adjacency List

An array of length n , such that each entry corresponds to a vertex v , and within each entry there is a list containing all of the vertices adjacent to v (aka all vertices that contribute to $out(v)$).



9.2.3 Graph Representation Comparison

	adjacency matrix	adjacency list
Space	$\Theta(n^2)$	$\Theta(n + m)$
Time to check if w adjacent to v	$\Theta(1)$	$\Theta(out(v))$
Time to visit all w adjacent to v .	$\Theta(n)$	$\Theta(out(v))$
Time to visit all edges	$\Theta(n^2)$	$\Theta(n + m)$

- **Checking if w is adjacent to v :**
 - **Adjacency Matrix:** just check if $A[v][w]$ is 1
 - **Adjacency List:** need to go over all vertices which are adjacent to v ; there are exactly $out(v)$ of them
- **Visit all w adjacent to v :**
 - **Adjacency Matrix:** need to go over all elements in the row $A[v]$. There are exactly n of them

- **Adjacency List:** need to go over all vertices which are adjacent to v ; there are exactly $out(v)$ of them
- **Visit all edges:**
 - **Adjacency Matrix:** need to go through the whole matrix, which has size n^2
 - **Adjacency List:** there are a total of $n + m$ elements stored in the list, which are all the vertices and all of their connections

9.2.4 Sparse and Dense Graphs

It must be the case that for a graph G :

$$m \leq n^2$$

- G is **dense** if $m \approx n^2$
- G is **sparse** if $m \ll n^2$

9.3 Traversing Graphs

- **Graph Traversal:** visiting all edges of the graph
- **General Traversal Strategy:** for any vertex v , visit all vertices reachable from v . Repeat for all unvisited vertices, until every single vertex has been visited

9.3.1 Breadth-First Search

- **Breadth-First Search:** traverse the graph in “layers”: visit v , visit $neighbours(v)$, visit $neighbours(neighbours(v))$, etc ...
 - we keep track of all visited nodes with a **visited** array
 - for every node $v \in V$, ensure that it hasn’t yet been visited
 - then, execute a breadth-first search strategy from v
 - * mark v as visited
 - * add v to a queue
 - * while the queue is non-empty, remove the first element, and add their neighbours to the queue (marking them as visited)
 - * by using the queue, we ensure to investigate all neighbours of a vertex before going deeper into the graph

Algorithm $\text{bfs}(G)$

1. Initialise Boolean array *visited*, setting all entries to FALSE.
2. Initialise *Queue* *Q*
3. **for all** $v \in V$ **do**
4. **if** *visited*[*v*] = FALSE **then**
5. bfsFromVertex(*G*, *v*)

Algorithm bfsFromVertex(G, v)

- ```

1. visited[v] = TRUE
2. Q.enqueue(v)
3. while not Q.isEmpty() do
4. u ← Q.dequeue()
5. for all w adjacent to u do
6. if visited[w] = FALSE then
7. visited[w] = TRUE
8. Q.enqueue(w)

```

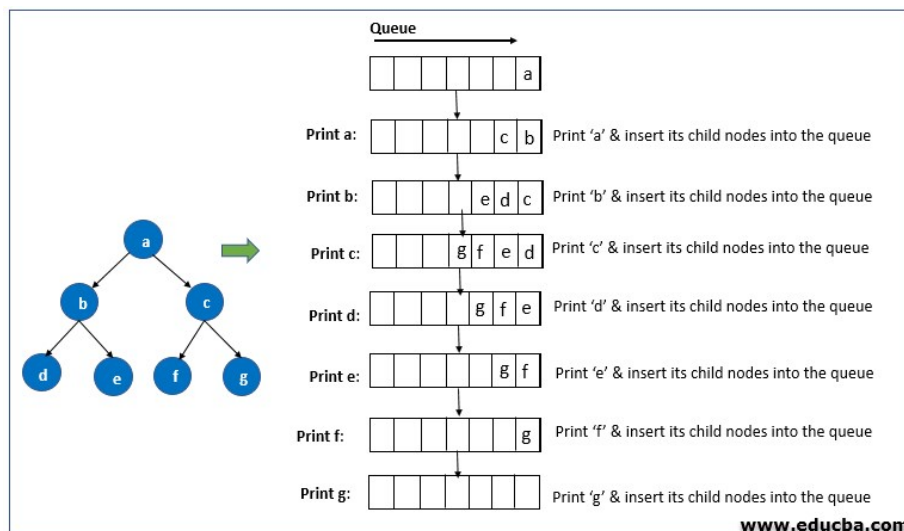


Figure 19: For a live demo, see Mary's videos (Week 9, Lecture 14, Part 3), they are **really** good

### 9.3.2 Depth-First Search

- **Depth-First Search:** traverse the graph by going as “deep” as possible: visit  $v$ , visit a neighbour  $w$  of  $v$ , visit a neighbour  $x$  of  $w$ , etc ...
  - we keep track of all visited nodes with a **visited** array
  - for every node  $v \in V$ , ensure that it hasn't yet been visited
  - then, execute a depth-first search strategy from  $v$ 
    - \* add  $v$  to a stack
    - \* while the stack isn't empty, remove the first item, and label it as visited. Then, add all of its adjacent nodes to the stack.
    - \* by using the stack, we ensure that the next item that we visit is the latest element that we added

#### Algorithm $\text{dfs}(G)$

1. Initialise Boolean array *visited*, setting all to FALSE
2. Initialise *Stack S*
3. **for all**  $v \in V$  **do**
4.     **if**  $\text{visited}[v] = \text{FALSE}$  **then**
5.          $\text{dfsFromVertex}(G, v)$

#### Algorithm $\text{dfsFromVertex}(G, v)$

1.  $S.\text{push}(v)$
2. **while not**  $S.\text{isEmpty}()$  **do**
3.      $u \leftarrow S.\text{pop}()$
4.     **if**  $\text{visited}[u] = \text{FALSE}$  **then**
5.          $\text{visited}[u] = \text{TRUE}$
6.         **for all**  $w$  adjacent to  $u$  **do**
7.              $S.\text{push}(w)$

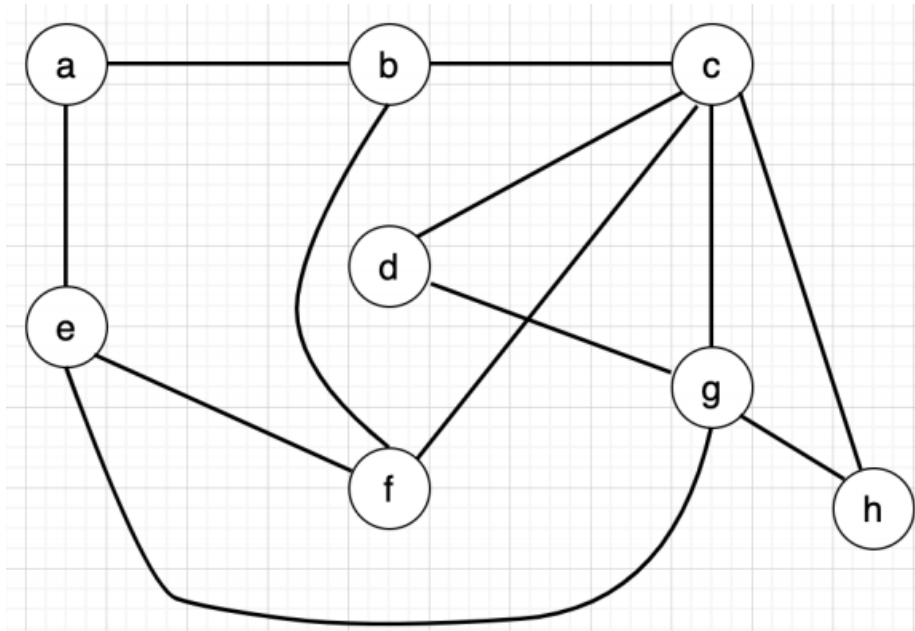


Figure 20: For a live demo, see Mary's videos (Week 9, Lecture 14, Part 5), they are **really** good

For the above graph:

- \*  $S = [a]$
- \* visit **a**,  $S = [e, b]$  (we add in lexicographic order, so b is added first)
- \* visit **e**,  $S = [g, f, a, b]$  (DFS doesn't worry about whether an item has been previously visited)
- \* visit **g**,  $S = [h, e, d, c, f, a, b]$
- \* visit **h**,  $S = [g, c, e, d, c, f, a, b]$
- \* g has already been visited, so just remove it;  $S = [c, e, d, c, f, a, b]$
- \* visit **c**,  $S = [h, g, f, d, b, e, d, c, f, a, b]$
- \* h has already been visited, so just remove it;  $S = [g, f, d, b, e, d, c, f, a, b]$
- \* g has already been visited, so just remove it;  $S = [f, d, b, e, d, c, f, a, b]$
- \* visit **f**,  $S = [e, c, b, d, b, e, d, c, f, a, b]$
- \* e has already been visited, so just remove it;  $S = [c, b, d, b, e, d, c, f, a, b]$
- \* c has already been visited, so just remove it;  $S = [b, d, b, e, d, c, f, a, b]$
- \* visit **b**,  $S = [f, c, a, b, d, b, e, d, c, f, a, b]$
- \* f has already been visited, so just remove it;  $S = [c, a, b, d, b, e, d, c, f, a, b]$
- \* c has already been visited, so just remove it;  $S = [a, b, d, b, e, d, c, f, a, b]$
- \* a has already been visited, so just remove it;  $S = [b, d, b, e, d, c, f, a, b]$

- \* b has already been visited, so just remove it;  $S = [d, b, e, d, c, f, a, b]$
- \* visit **d**,  $S = [g, c, b, e, d, c, f, a, b]$
- \* by this time, all elements have been visited, so we will just pop all items and get an empty stack, so DFS terminates

### 9.3.3 Recursive Depth-First Search

- **Recursive Depth-First Search:** DFS is well suited for recursion, as at each node we just want to go as deep as possible. We can thus adapt DFS to be recursive, and not require a stack.

#### Algorithm `dfs(G)`

1. Initialise Boolean array *visited*, setting all to FALSE
2. **for all**  $v \in V$  **do**
3.     **if** *visited*[ $v$ ] = FALSE **then**
4.         `dfsFromVertex(G, v)`

#### Algorithm `dfsFromVertex(G, v)`

1. *visited*[ $v$ ]  $\leftarrow$  TRUE
2. **for all**  $w$  adjacent to  $v$  **do**
3.     **if** *visited*[ $w$ ] = FALSE **then**
4.         `dfsFromVertex(G, w)`

### 9.3.4 Runtime Analysis of Traversal Strategies

Recursive DFS allows us to derive a runtime for DFS. Moreover, the runtime of DFS and BFS will be identical (think that we are traversing the same nodes, but in different order; there can be additional work depending on the graph, but that will just be  $\Theta(1)$ )

- **dfsFromVertex is Called Only Once:** it is called at least once:
  - either it is called in recursive step, which eventually sets *visited*( $v$ ) = TRUE
  - or it is called after the for loop if *visited*( $v$ ) = FALSE

and it is called at most once, as once *visited*( $v$ ) = TRUE, `dfsFromVertex` can never be called again on  $v$

- if a graph is directed,  $\sum_{v \in V} \text{out}(v) = m$
- if a graph is undirected,  $\sum_{v \in V} \text{out}(v) = 2m$
- **Runtime of Recursive DFS:** `dfs` iterates over all vertices, which takes  $\Theta(n)$  time. Then, in calling `dfsFromVertex`, for each vertex  $v$  it does

$\Theta(out(v))$  work, assuming that we use an adjacency list (eventually visits all adjacent nodes, and excluding recursion). Thus, runtime is:

$$\begin{aligned} T(m, n) &= \Theta(n) + \sum_{v \in V} \Theta(out(v)) \\ &= \Theta(n + m) \end{aligned}$$

- **Recursive DFS vs Iterative DFS:** these 2 methods are essentially identical, with iterative using a stack and while loop. But the work with the stack is bounded by  $n$ , so the runtime will be identical (see Lecture 15, Slide 7)

## 9.4 DFS Forests

- **Tree:** an undirected graph, in which any 2 vertices are connected by at most 1 edge
- **Forest:** a collection of trees
- **DFS Forest:** traversing a graph via DFS can build up a forest
  - $w$  is a child of  $v$  in a DFS forest if after calling `dfsFromVertex(G, v)` we call `dfsFromVertex(G, v)`

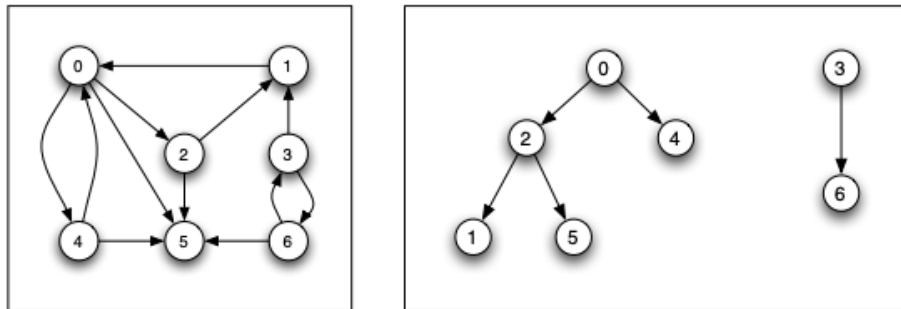


Figure 21: DFS Forests will vary depending on the start element with which we begin `dfs`

## 9.5 Topological Ordering and TopologicalSort

### 9.5.1 Topological Order

- $\prec$ : used to define an ordering of 2 elements
  - for example, if Task 0 must be completed before Task 2, we could write  $0 \prec 2$

- **Total Order of a Set:** given a set, every element is related to some other element via  $\prec$
- **Topological Order of a Graph:** given a **directed graph**, its topological order is a total ordering of  $V$ , such that, for any edge  $(v, w) \in E$ ,  $v \prec w$ 
  - if there is an edge going from  $v$  to  $w$ , then any topological ordering must ensure that  $v \prec w$  (i.e  $v$  is traversed before  $w$ )

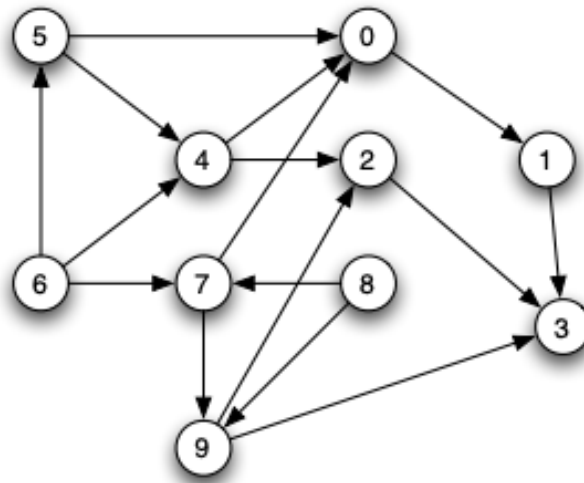


Figure 22: A potential topological ordering for this graph is  $8 \prec 6 \prec 7 \prec 9 \prec 5 \prec 4 \prec 2 \prec 0 \prec 1 \prec 3$

### 9.5.2 Theorems

Recursive DFS can be used to determine a potential topological order for a graph. For that, we develop the following theorems and notation.

- **Directed Acyclic Graph:** a digraph (directed graph) without cycles
- *A digraph has a topological order **if and only if** it is a DAG*
- **Finished Vertex:**  $v$  is finished if `dfsFromVertex(G,v)` has finished recursing
- **Vertex Classification From Recursive DFS:**
  - *white*: vertex which hasn't been visited
  - *grey*: vertex which has been visited but isn't finished
  - *black*: vertex which is finished

- Let  $v$  be a vertex, and we have began executing  $\text{dfsFromVertex}(G, v)$ . For any vertex  $w$ :
  - if a vertex  $w$  is white and reachable from  $v$ , then  $w$  will be black before  $v$ 
    - \* we require  $\text{dfsFromVertex}(G, w)$  to terminate in order for  $\text{dfsFromVertex}(G, v)$  to terminate
  - if  $w$  is grey,  $v$  is reachable from  $w$ 
    - \*  $w$  being grey before  $\text{dfsFromVertex}(G, v)$  was called means that there has been a previous call which reached  $w$ . Since  $w$  is not yet black, this call must still be going, so in particular,  $\text{dfsFromVertex}(G, w)$  must have been called, and this must have happened before  $\text{dfsFromVertex}(G, v)$ . In other words, if we find a grey node, there must be a cycle, so no topological order is possible.
- define an order:

$$v \prec w \iff w \text{ is black before } v$$

if  $G$  is a DAG,  $\prec$  defines a topological order for  $G$

- to see this, consider calling  $\text{dfsFromVertex}(G, v)$ , and let  $(v, w) \in E$ . Then:
  - \*  $w$  already black means  $v \prec w$
  - \*  $w$  is white, but by above we know that eventually  $w$  will be black before  $v$
  - \*  $w$  is grey, but this implies that  $G$  has a cycle (by above), so contradiction

### 9.5.3 Topological Sort

- **Topological Sort:** a use of DFS to derive a topological sorting for a graph
  - all vertices begin coloured as white
  - for any vertex  $v$ , label it grey and apply **sortFromVertex** (basically recursive DFS) to all of its adjacent vertices **which are white**
  - when we finish iterating over all adjacents, label  $v$  as black
  - in essence, when going depth first, we eventually reach a node which has no neighbours, and we label it as black

**Algorithm** topSort( $G$ )

1. Initialise array *state*  
by setting all entries to *white*.
2. Initialise linked list  $L$
3. **for all**  $v \in V$  **do**
4.     **if**  $state[v] = white$  **then**
5.         sortFromVertex( $G, v$ )
6. **print**  $L$

**Algorithm** sortFromVertex( $G, v$ )

1.  $state[v] \leftarrow grey$
2. **for all**  $w$  adjacent to  $v$  **do**
3.     **if**  $state[w] = white$  **then**
4.         sortFromVertex( $G, w$ )
5.     **else if**  $state[w] = grey$  **then**
6.         **print** "G has a cycle"
7.     **halt**
8.  $state[v] \leftarrow black$
9.  $L.insertFirst(v)$

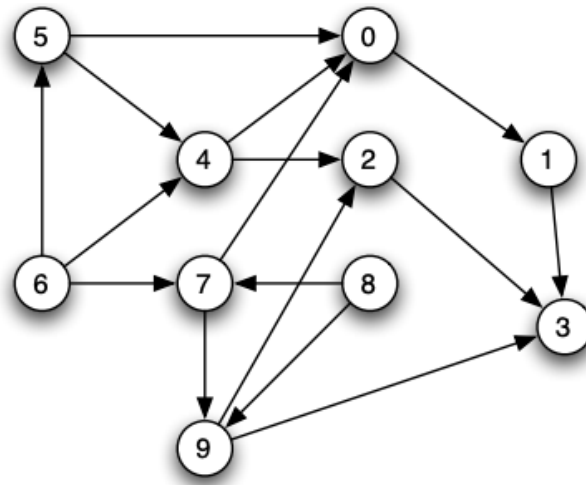


Figure 23: Again, Mary's lecture is really amazing for this (Lecture 15, Part 4

- For the above DAG:

- we start at 0, it is white, so `sortFromVertex(G,0)`
  - \* make 0 grey
  - \* only 1 is adjacent, so use `sortFromVertex(G,1)`
  - \* make 1 grey
  - \* only 3 is adjacent, so use `sortFromVertex(G,3)`
  - \* make 3 grey
  - \* since 3 has no adjacent nodes, we miss the for loop, and make 3 black
  - \* we have now recursed over all adjacent nodes to 1, so we exit the for loop, and make 1 black
  - \* we have now recursed over all adjacent nodes to 0, so we exit the for loop, and make 0 black
- the above would give us a (partial) ordering of  $0 \prec 1 \prec 3$
- then we do the same, but for the remaining nodes
- **TopSort Runtime:** since it is essentially DFS, `TopSort` has  $\Theta(n + m)$  runtime

## 9.6 Connected Components

- **Connected Vertices:** a subset  $C \subseteq V$  is connected if  $\forall v, w \in C$ , we can find a path from  $v$  to  $w$ 
  - **strongly connected** if the graph is directed
- **Connected Component of a Graph:** maximum connected subset  $C$  of  $V$  (a subgraph in which every node is connected)
- **Connected Graph:** a graph with only 1 connected component (all vertices are connected by a path)
- **Connected Components in Undirected Graphs:**
  - a vertex  $v$  is in the connected component of all the nodes reachable from  $v$
  - each vertex  $v$  is in exactly one connected component (connected components are disjoint)
  - `dfsFromVertex` or `bfsFromVertex` visit all elements in the connected component of a given vertex

**Algorithm** connComp( $G$ )

1. Initialise Boolean array *visited*  
by setting all entries to FALSE
2. **for all**  $v \in V$  **do**
3.     **if** *visited*[ $v$ ] = FALSE **then**
4.         **print** "New Component"
5.         ccFromVertex( $G, v$ )

**Algorithm** ccFromVertex( $G, v$ )

1. *visited*[ $v$ ]  $\leftarrow$  TRUE
2. **print**  $v$
3. **for all**  $w$  adjacent to  $v$  **do**
4.     **if** *visited*[ $w$ ] = FALSE **then**
5.         ccFromVertex( $G, w$ )

Figure 24: The algorithm is essentially Recursive DFS