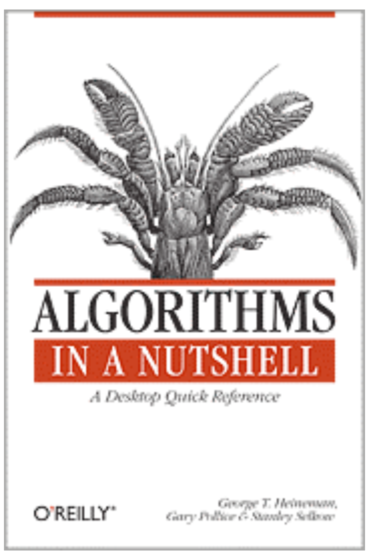
# Algorithms in a Nutshell




Session 4

Recap Algorithm Themes

11:20 – 11:40

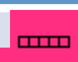
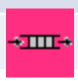
# Outline

- Data Structures
  - Array, Linked List, Queue, Heap, Priority Queue, Tree, Graph
- Space vs. Time tradeoff
- Approaches
  - Divide and conquer
  - Greedy algorithm

# Common Data Structures

- Basic Structures

Indexed access (Get $i^{th}$, Set $i^{th}$)

| Structure | Glyph | Insert | Delete | Get $i^{th}$ | Set $i^{th}$ | Find |
|---|---|---|---|---|---|---|
| Array |  | O(n) | O(n) | O(1) | O(1) | O(n) |
| Linked List |  | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack |  | O(1) | O(1) | ---- | ---- | ---- |
| Queue |  | O(1) | O(1) | ---- | ---- | ---- |

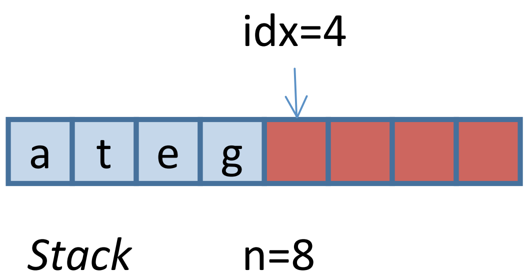
<u>Linked List</u>
insert adds to front
insert adds to tail
remove from any location

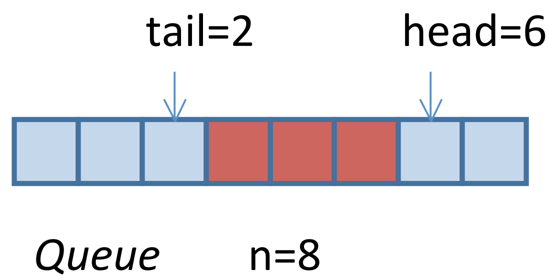
<u>Stack</u>
insert is push
remove is pop

<u>Queue</u>
Insert adds to one end
remove extracts from other end

# Dynamic vs. Static sizes

- Fixed size allocation via arrays



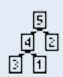



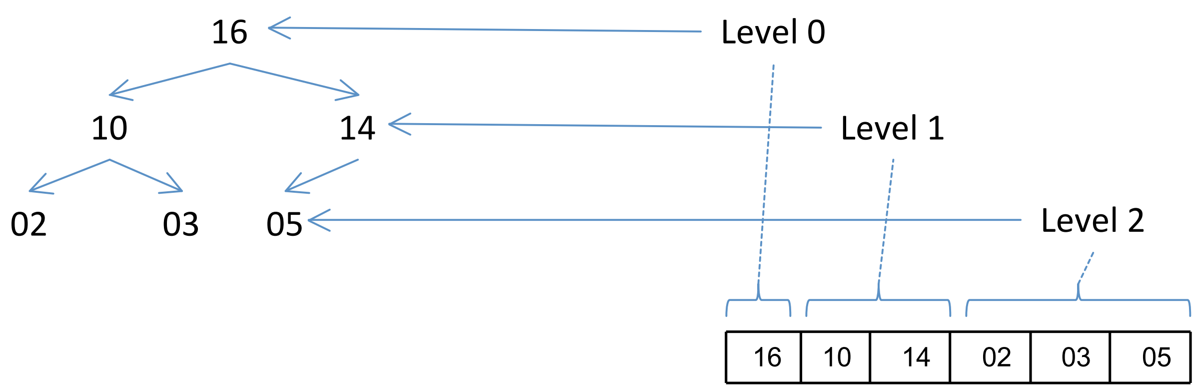

- Increase size by allocating more memory
  - Don’t increase by fixed amount, but double
  - If you only add linear amount each time, too inefficient

```
int oldCapacity = table.length;
Entry[] oldMap = table;

int newCapacity = oldCapacity * 2 + 1;
Entry[] newMap = new Entry[newCapacity];

table = newMap;
...
```

# Binary Heap

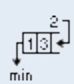
- Heap can be stored in array
  - Fixed maximum size
  - Assumes you only remove elements

| Structure | Glyph | Insert | Remove Max | Find |
|---|---|---|---|---|
| Binary Heap |  | O(log n) | O(log n) | -- |



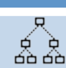
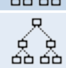
# Priority Queue

- Most implementations provide only
  - `insert (element, priority)`
  - `getMinimum()`
- If you only need these two operations, Binary Heap can be used
- Often need one more method
  - `decreaseKey (element, newPriority)`
  - If you need this one also, you must adjust data structure

| Structure | Glyph | Insert | Remove Max | Contains | DecreaseKey |
|---|---|---|---|---|---|
| Priority Queue |  | O(log n) | O(log n) | O(log n) | O(log n) |

# Balanced Binary Tree

- Ideal dynamic data structure
  - No need to know maximum size in advance
  - Red/Black Tree implementations quite common
- Avoids worst case behavior
  - Which might degenerate to O(n) for all operations

| Structure | Glyph | Insert | Delete | Find |
|---|---|---|---|---|
| Tree |  | O(log n) | O(log n) | O(log n) |
| Balanced Binary Tree |  | O(log n) | O(log n) | O(log n) |

# Implementation Tradeoff

- Algorithm designers have developed innovative data structures
  - Fibonacci Heaps
  - Skip lists
  - Splay trees
- Theoretical improvement is offset by more complicated implementations
  - Also improvement is “amortized” over life of use
  - Some operations may be worse than expected

# Divide and Conquer

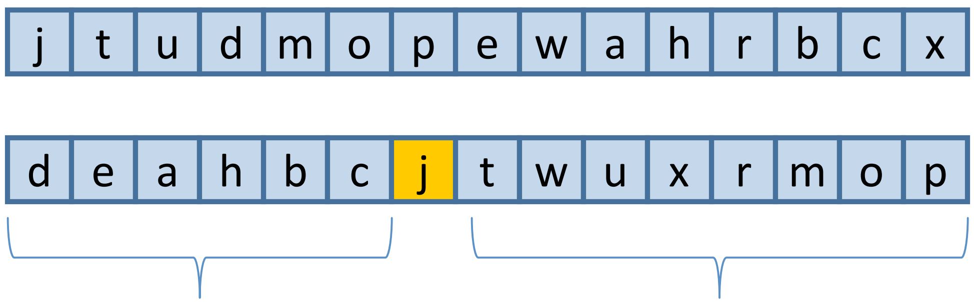
- Intuition why it works so well
  - Look for word in Dictionary
  - Each iteration discards half of remaining words during search
- Number of iterations
  - $\log_2 n = \log n$ throughout book
  - O(log n) family
- Clearly much better than linear scan of n elements

| Words to search |
|---|
| 1,048,576 |
| 524,288 |
| 262,144 |
| 131,072 |
| 65,536 |
| 32,768 |
| 16,384 |
| 8,192 |
| 4,096 |
| 2,048 |
| 1,024 |
| 512 |
| 256 |
| 128 |
| 64 |
| 32 |
| 16 |
| 8 |
| 4 |
| 2 |
| 1 |

30

# Divide and Conquer

- Also applies to composed problems
  - QUICKSORT




SortTime(15)

SortTime(6) + SortTime(8)

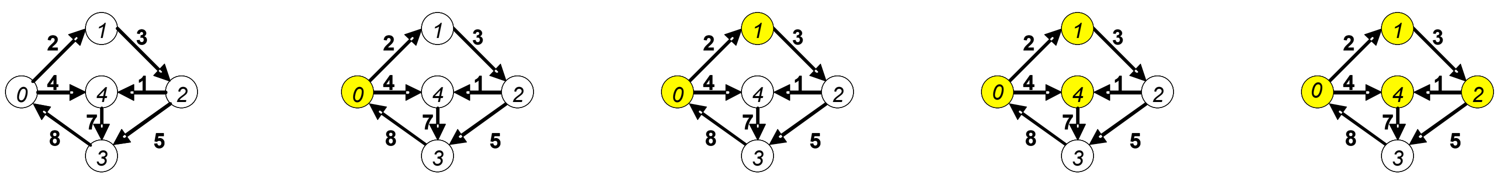
SortTime(15) = TimePartition(15) +SortTime(6) + SortTime(8)

$T(n) = O(n) + 2*T(n/2)$
$T(n) = 2*O(n) + 4*T(n/4)$
$T(n) = 3*O(n) + 8*T(n/8)$
$T(n) = k*O(n) + 2^k*T(n/2^k)$

Continues k=log n times
$T(n) = \log n*O(n) + O(n)$
$T(n) = O(n * \log n)$

# Greedy Algorithm

- Goal is to solve problem of size *n*
  - Single-Source Shortest Path from *s* to all vertices $v_i$
  - DIJKSTRA'S Algorithm
- Make locally optimal decision at each stage
  - Apply until result yields globally optimal solution




| | | | | | |
|---|---|---|---|---|---|
| dist | 0 | ∞ | ∞ | ∞ | ∞ |
| visited | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| dist | 0 | 2 | ∞ | ∞ | 4 |
| visited | ✓ | | | | |

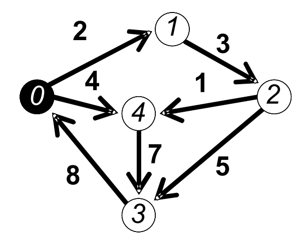
| | | | | | |
|---|---|---|---|---|---|
| dist | 0 | 2 | 5 | ∞ | 4 |
| visited | ✓ | ✓ | | | |

| | | | | | |
|---|---|---|---|---|---|
| dist | 0 | 2 | 5 | 11 | 4 |
| visited | ✓ | ✓ | | | ✓ |

| | | | | | |
|---|---|---|---|---|---|
| dist | 0 | 2 | 5 | 10 | 4 |
| visited | ✓ | ✓ | ✓ | | ✓ |

# Dynamic Programming

- Goal is to solve problem of size *n*
  - All Pairs Shortest Path between any vertices ($v_i$, $v_j$)
  - FLOYD-WARSHALL Algorithm
- Solve most constrained problems first
  - Relax constraints systematically until done



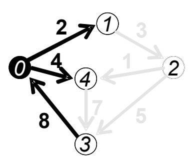

*Shortest distance considering just initial edges*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | ∞ | ∞ | 4 |
| 1 | ∞ | 0 | 3 | ∞ | ∞ |
| 2 | ∞ | ∞ | 0 | 5 | 1 |
| 3 | 8 | ∞ | ∞ | 0 | ∞ |
| 4 | ∞ | ∞ | ∞ | 7 | 0 |

**dist[u][v]**

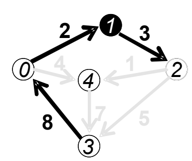
*Shortest path can now include vertex 0*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | ∞ | ∞ | 4 |
| 1 | ∞ | 0 | 3 | ∞ | ∞ |
| 2 | ∞ | ∞ | 0 | 5 | 1 |
| 3 | 8 | 10 | ∞ | 0 | 12 |
| 4 | ∞ | ∞ | ∞ | 7 | 0 |



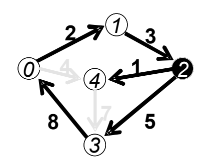
*Shortest path can now include vertices 0 + 1*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | ∞ | 4 |
| 1 | ∞ | 0 | 3 | ∞ | ∞ |
| 2 | ∞ | ∞ | 0 | 5 | 1 |
| 3 | 8 | 10 | 13 | 0 | 12 |
| 4 | ∞ | ∞ | ∞ | 7 | 0 |



*Shortest path can now include vertices 0 + 1 + 2*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | 10 | 4 |
| 1 | ∞ | 0 | 3 | 8 | 4 |
| 2 | ∞ | ∞ | 0 | 5 | 1 |
| 3 | 8 | 10 | 13 | 0 | 12 |
| 4 | ∞ | ∞ | ∞ | 7 | 0 |



*Final result shown below*

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 5 | 10 | 4 |
| 1 | 16 | 0 | 3 | 8 | 4 |
| 2 | 13 | 15 | 0 | 5 | 1 |
| 3 | 8 | 10 | 13 | 0 | 12 |
| 4 | 15 | 17 | 20 | 7 | 0 |

# Summary

- Various data structures investigated
- Various approaches described
  - Divide and conquer
  - Greedy algorithm
  - Dynamic programming