SCC.121: Fundamentals of Computer Science

Revision

Sorting Algorithms

	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Best case	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Average case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Worst case	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
In-place	Yes	Yes	No	Yes

Sorting Algorithms

	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Best case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(n)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$
Average case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$
Worst case	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n}^2)$	$\mathrm{O}(\mathrm{n} \log \mathrm{n})$	$\mathrm{O}(\mathrm{n}^2)$
In-place	Yes	Yes	No	Yes

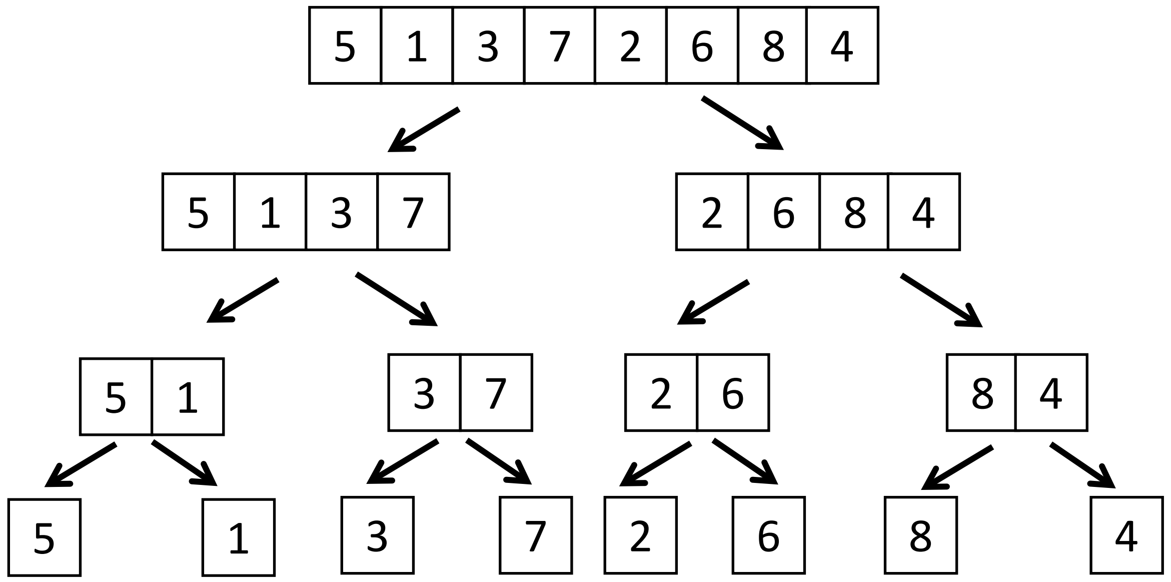
Recursive MergeSort

1. Repeatedly cut up your initial array into “equal” halves,
2. Merge the size 1 arrays back into sorted size 2 arrays, and so on…

Recursive MergeSort

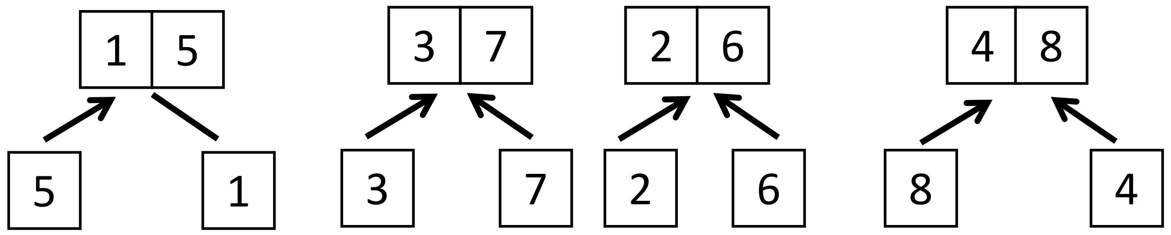
1. **Repeatedly cut up your initial array into "equal" halves,**
2. Merge the size 1 arrays back into sorted size 2 arrays, and so on…

(Recursion Tree, depth 3)



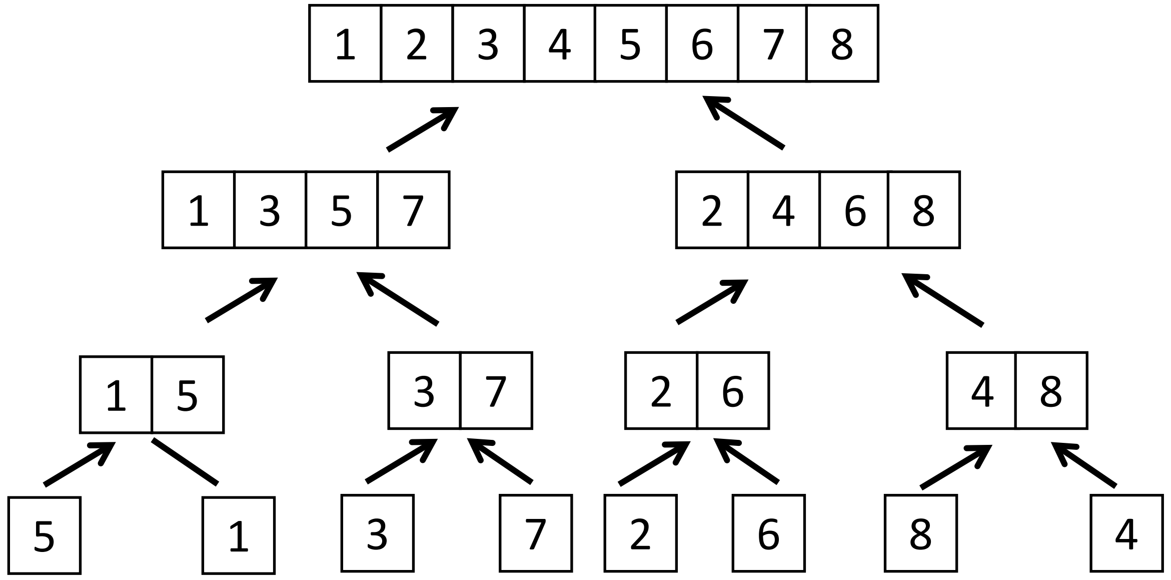
Recursive MergeSort

1. Repeatedly cut up your initial array into “equal” halves,
2. **Merge the size 1 arrays back into sorted size 2 arrays, and so on…**



Recursive MergeSort

1. Repeatedly cut up your initial array into “equal” halves,
2. **Merge the size 1 arrays back into sorted size 2 arrays, and so on…**



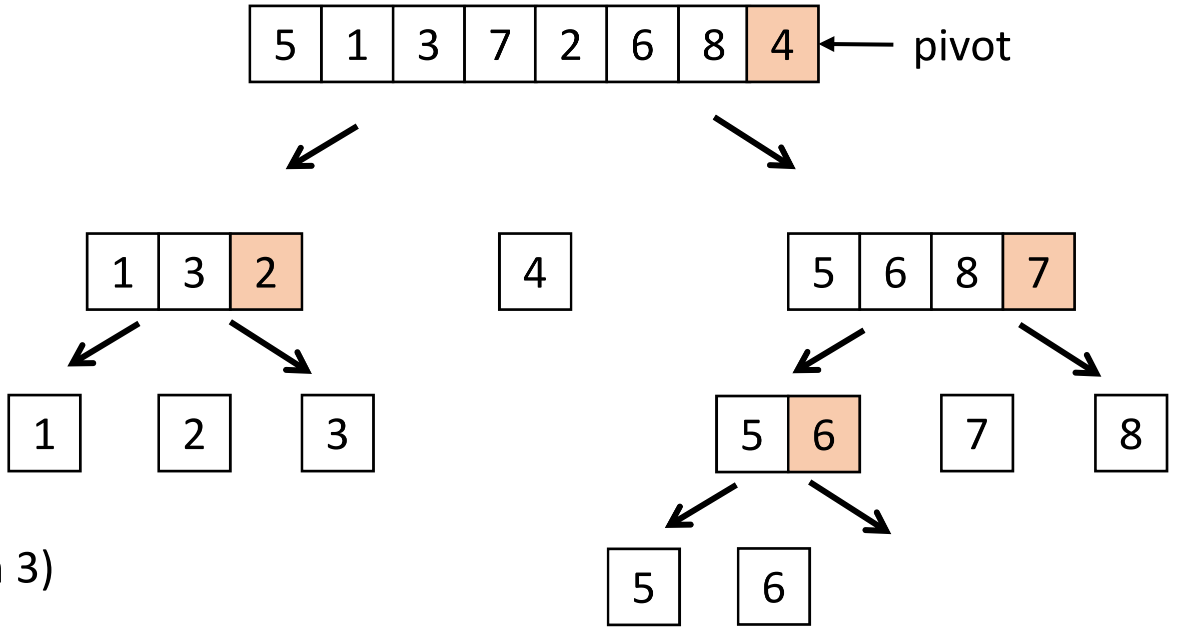
Recursive Quicksort

1. From your initial array, do as follows:
    1. Choose a **pivot** value,
    2. Divide array into three subarrays
        1. One with all values strictly smaller to the pivot,
        2. One with the pivot value only
        3. One with all values greater or equal to the pivot
    3. Recurse on the first and third sub-array.

2. Concatenate the (now sorted) sub-arrays

Recursive (In-Place) Quicksort

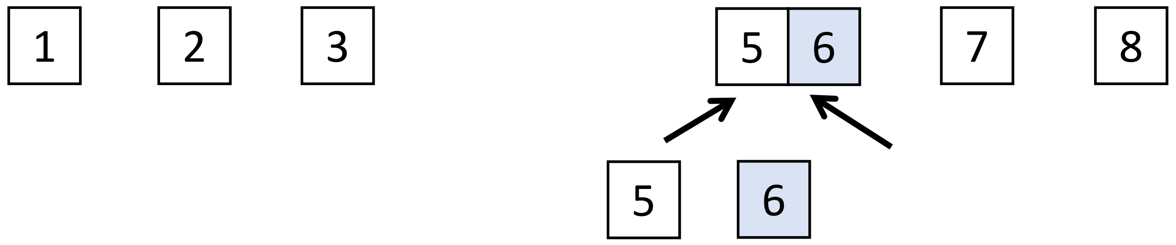
1. **Divide array into two according to pivot (see code in week 16)**
2. Concatenate the (now sorted) sub-arrays



(Recursion Tree, depth 3)

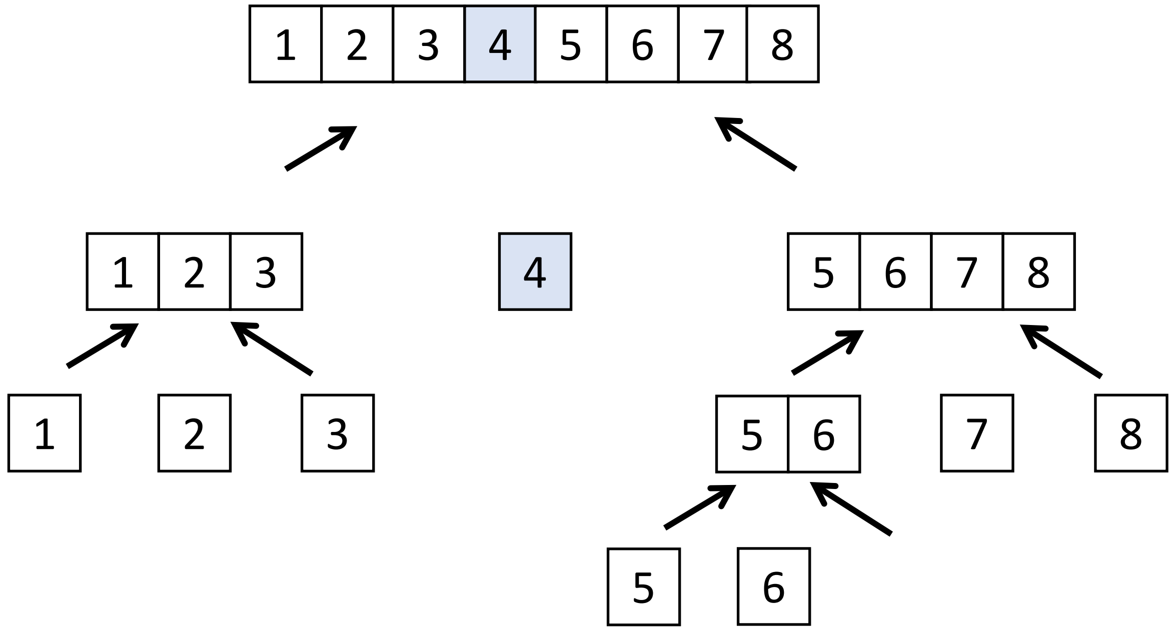
Recursive (In-Place) Quicksort

1. Divide array into two according to pivot (see code in week 16)
2. **Concatenate the (now sorted) sub-arrays**



Recursive (In-Place) Quicksort

1. Divide array into two according to pivot (see code in week 16)
2. **Concatenate the (now sorted) sub-arrays**



Dictionary ADT and BSTs

- Dictionary (or Map) abstract data structure allows for the insertion, deletion and search of <key,value> pairs.
- One way to implement the Dictionary ADT is using Binary Search Trees (BSTs):

- <u>Binary search tree:</u>
  - Binary tree
  - Each node holds a <key,value> pair
  - Nodes are "sorted" by their keys. For example, for a node with key 17, all nodes in its left subtree contain keys that are strictly smaller than 17, and all nodes in its right subtree contain keys that are greater or equal to 17.

Dictionary ADT and BSTs

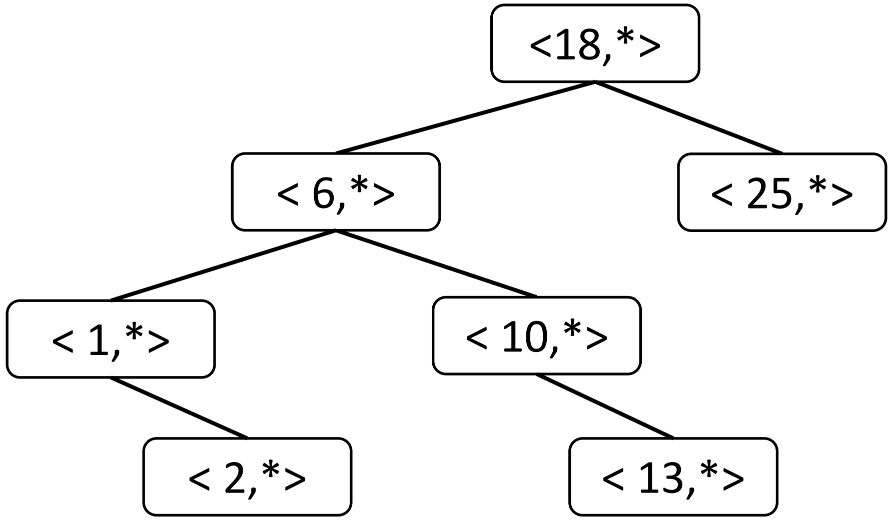
Naïvely inserting <key,value> pairs in a BST

- <18,*>, < 6,*>, < 10,*>, < 25,*>, < 1,*>, < 13,*>, < 2,*>,

Dictionary ADT and BSTs

Naïvely inserting <key,value> pairs in a BST

- <18,*>, < 6,*>, < 10,*>, < 25,*>, < 1,*>, < 13,*>, < 2,*>,



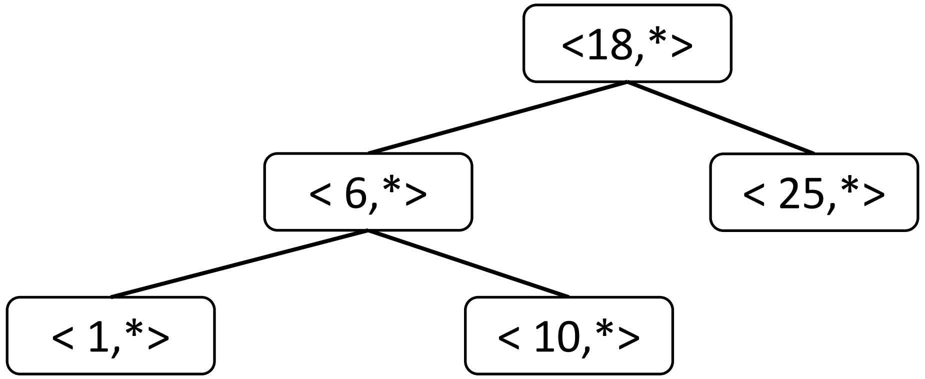
Unbalanced tree:

- Subtree rooted at < 25,*> has depth 0
- Subtree rooted at < 6,*> has depth 2

Dictionary ADT and BSTs

Naïvely inserting <key,value> pairs in a BST

- <18,*>, < 6,*>, < 10,*>, < 25,*>, < 1,*>



Balanced tree:

- Subtree rooted at < 25,*> has depth 0
- Subtree rooted at < 6,*> has depth 1

Dictionary ADT and BSTs

- Search takes $O(h)$ time on BSTs, where h is the height of the tree
- Naïve insertion and deletion also take $O(h)$ time, and do not maintain the tree balanced

- Wide efficiency gap of search between
  - **Unbalanced** BSTs with heights **that can be** $\boldsymbol{h = \Theta(n)}$
  - **Balanced** BSTs with **heights that must be** $\boldsymbol{h = \Theta(\log n)}$

- In practice (and available in java collections, for example), you can implement a **self-balancing binary search tree data structure**.

Graphs

Adjacency Matrix:

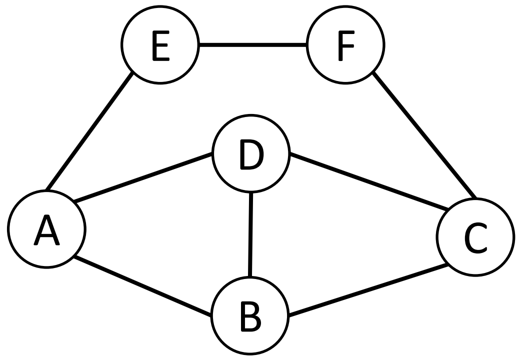
$$\begin{array}{c|c|c|c|c|c|c|}
\text{(A)} & 0 & 1 & 0 & 1 & 1 & 0 \\
\text{(B)} & 1 & 0 & 1 & 1 & 0 & 0 \\
\text{(C)} & 0 & 1 & 0 & 1 & 0 & 1 \\
\text{(D)} & 1 & 1 & 1 & 0 & 0 & 0 \\
\text{(E)} & 1 & 0 & 0 & 0 & 0 & 1 \\
\text{(F)} & 0 & 0 & 1 & 0 & 1 & 0
\end{array}$$

Adjacency List:

A:	B D E
B:	A C D
C:	B D F
D:	A B C
E:	A F
F:	C E

Graphs

Revision of basic definitions for unweighted, undirected graphs:

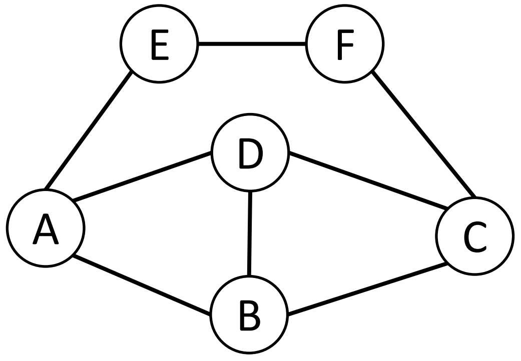


- $Density \; = \frac{2|E|}{|V|(|V|-1)}$
- $Distance(u, v) =$ number of edges in the shortest path between u and v
- $Diameter \; = \max_{u,v \in V} Distance(u, v)$

Graphs

Revision of basic definitions for unweighted, undirected graphs:

$$\begin{array}{c|c|c|c|c|c|c|}
\text{(A)} & 0 & 1 & 0 & 1 & 1 & 0 \\
\text{(B)} & 1 & 0 & 1 & 1 & 0 & 0 \\
\text{(C)} & 0 & 1 & 0 & 1 & 0 & 1 \\
\text{(D)} & 1 & 1 & 1 & 0 & 0 & 0 \\
\text{(E)} & 1 & 0 & 0 & 0 & 0 & 1 \\
\text{(F)} & 0 & 0 & 1 & 0 & 1 & 0
\end{array}$$

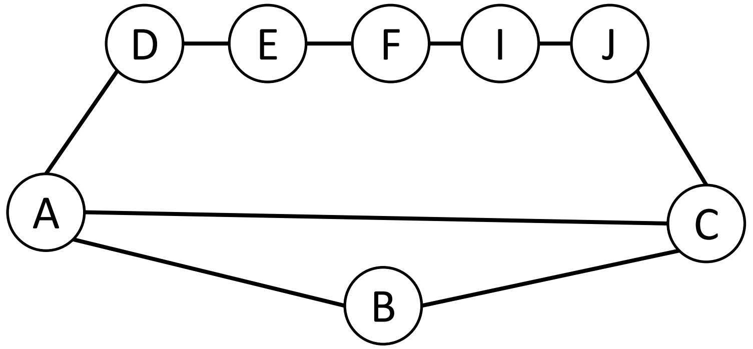


- $Density \ = \frac{2*8}{6*5} = \frac{8}{15}$
- $Distance(A, C) = 2$
- $Diameter \ = 2$

Graphs

Revision of basic definitions for unweighted, undirected graphs:

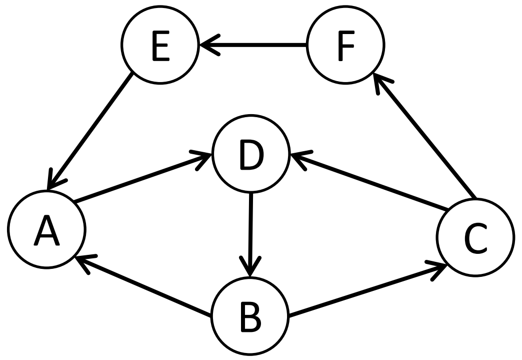
(A)	0	1	1	1	0	0	0	0
(B)	1	0	1	0	0	0	0	0
(C)	1	1	0	0	0	0	0	1
(D)	1	0	0	0	1	0	0	0
(E)	0	0	0	1	0	1	0	0
(F)	0	0	0	0	1	0	1	0
(I)	0	0	0	0	0	1	0	1
(J)	0	0	1	0	0	0	1	0



- $Density \;= \frac{2*9}{8*7} = \frac{9}{28}$
- $Distance(A, C) = 1$
- $Distance\ (D, J)\ =\ 3$
- $Diameter\ = 4$

Graphs

Revision of basic definitions for unweighted, directed graphs:

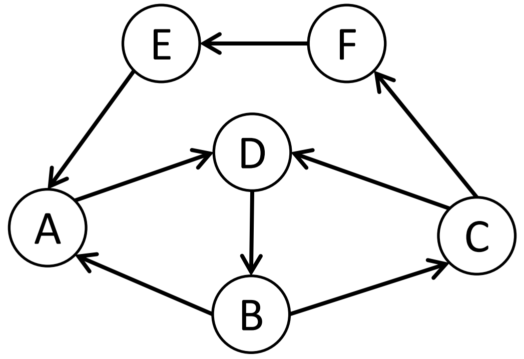


- $Density = \frac{|E|}{|V|(|V|-1)}$
- $Distance(u, v)$ = number of edges in the shortest directed path between u and v
- $Diameter = \max_{u,v \in V} Distance(u, v)$

Graphs

Revision of basic definitions for unweighted, directed graphs:

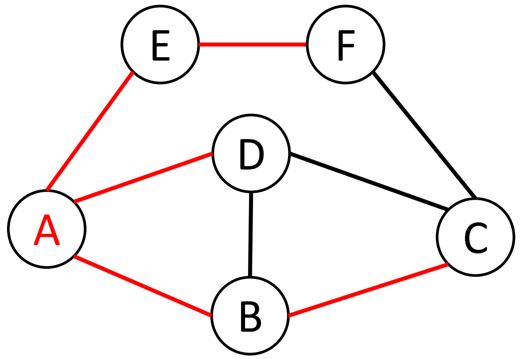
$$\begin{array}{c|cccccc} (A) & 0 & 0 & 0 & 1 & 0 & 0 \\ (B) & 1 & 0 & 1 & 0 & 0 & 0 \\ (C) & 0 & 0 & 0 & 1 & 0 & 1 \\ (D) & 0 & 1 & 0 & 0 & 0 & 0 \\ (E) & 1 & 0 & 0 & 0 & 0 & 0 \\ (F) & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$



- $Density \; = \frac{8}{6*5} = \frac{4}{15}$
- $Distance(A, C) = 3$
- $Diameter \; = 5$

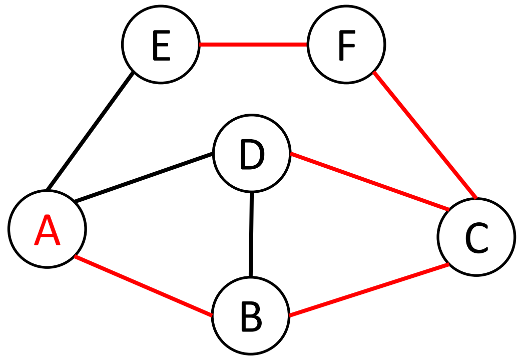
Graph Searches

BFS started at A:



Depth=2

DFS started at A:

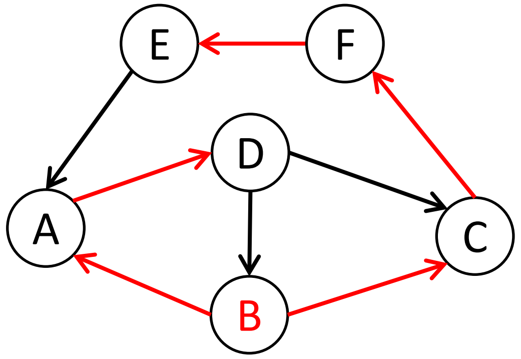


Depth=4

In these graph searches, whenever multiple edges can be chosen, the edge which is **lexicographically first** is taken.

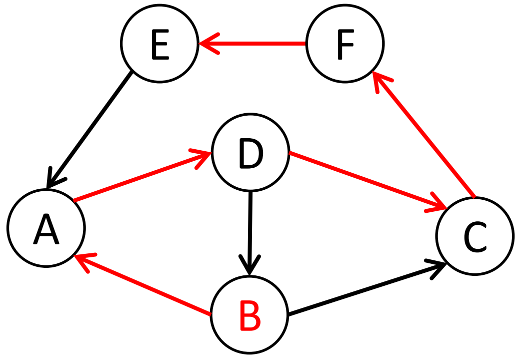
Graph Searches

BFS started at A:



Depth=3

DFS started at A:

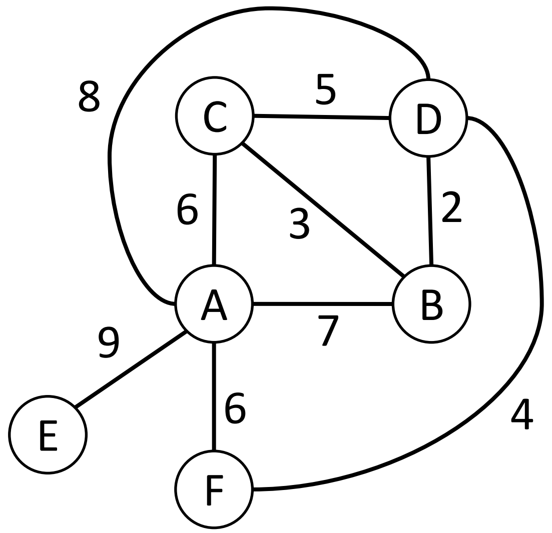


Depth=5

(Note that this directed graph is slightly different from the one two slides ago. In that other graph, BFS and DFS searches started at A give the same tree.)

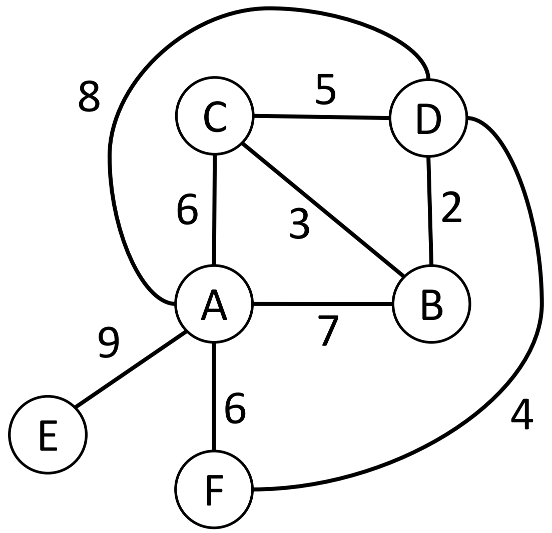
Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:

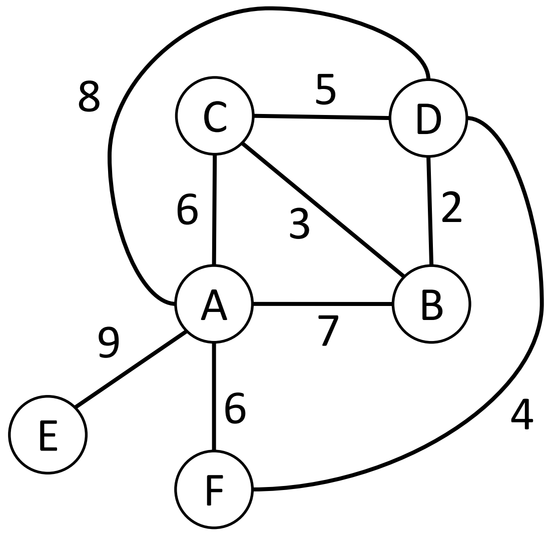


Method 1 (Kruskal):

Iterate through edges in order of increasing weights, and add edge if it does not create a cycle

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



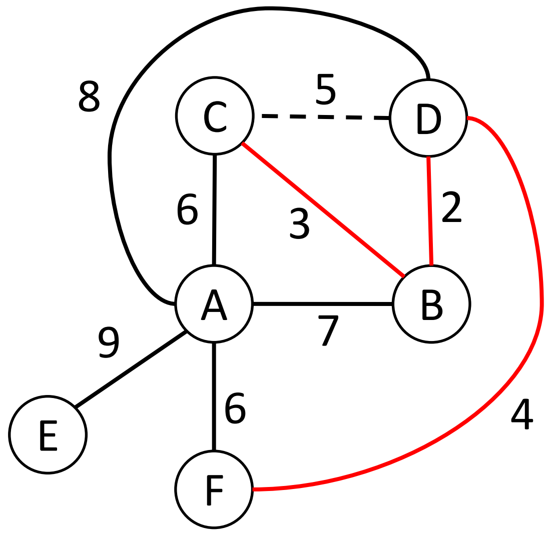
Method 1 (Kruskal):

Iterate through edges in order of increasing weights, and add edge if it does not create a cycle

1. BD
2. BC
3. DF
4. CD
5. AC
6. AF
7. AB
8. AD
9. AE

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



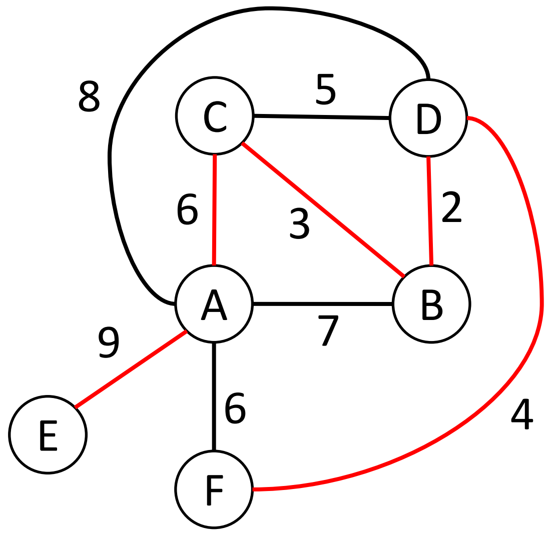
Method 1 (Kruskal):

Iterate through edges in order of increasing weights, and add edge if it does not create a cycle

1. BD
2. BC
3. DF
4. ~~CD~~
5. AC
6. AF
7. AB
8. AD
9. AE

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



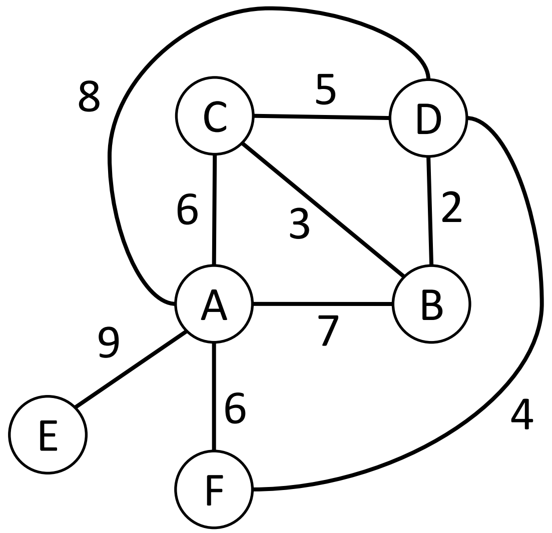
Method 1 (Kruskal):

Iterate through edges in order of increasing weights, and add edge if it does not create a cycle

1. BD
2. BC
3. DF
4. ~~CD~~
5. AC
6. ~~AF~~
7. ~~AB~~
8. ~~AD~~
9. AE

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:

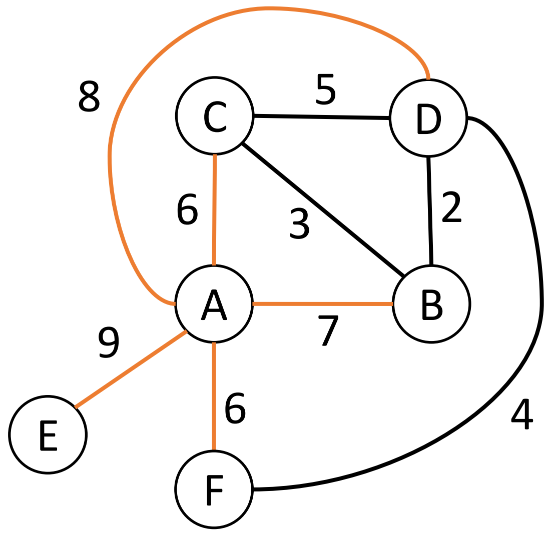


Method 2 (Prim):

Start at a node, and repeatedly add incident outgoing edge of minimum weight

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



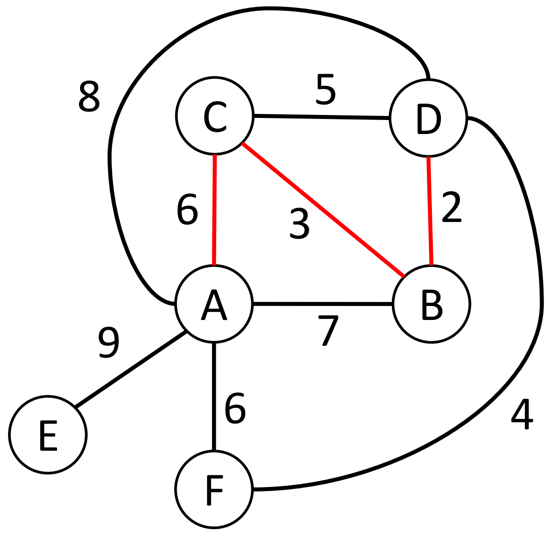
Method 2 (Prim):

Start at a node, and repeatedly add incident outgoing edge of minimum weight

1. A

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



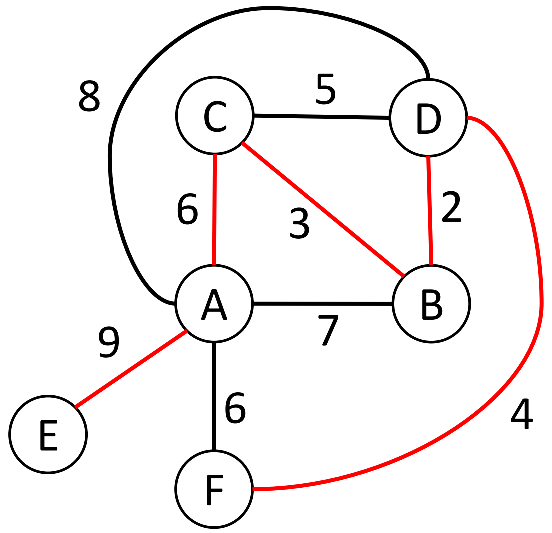
Method 2 (Prim):

Start at a node, and repeatedly add incident outgoing edge of minimum weight

1. A
2. AC
3. BC
4. BD

Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



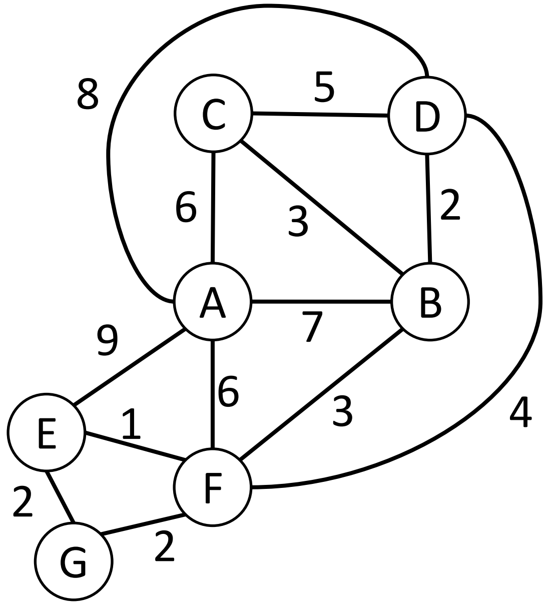
Method 2 (Prim):

Start at a node, and repeatedly add incident outgoing edge of minimum weight

1. A
2. AC
3. BC
4. BD
5. DF
6. AE

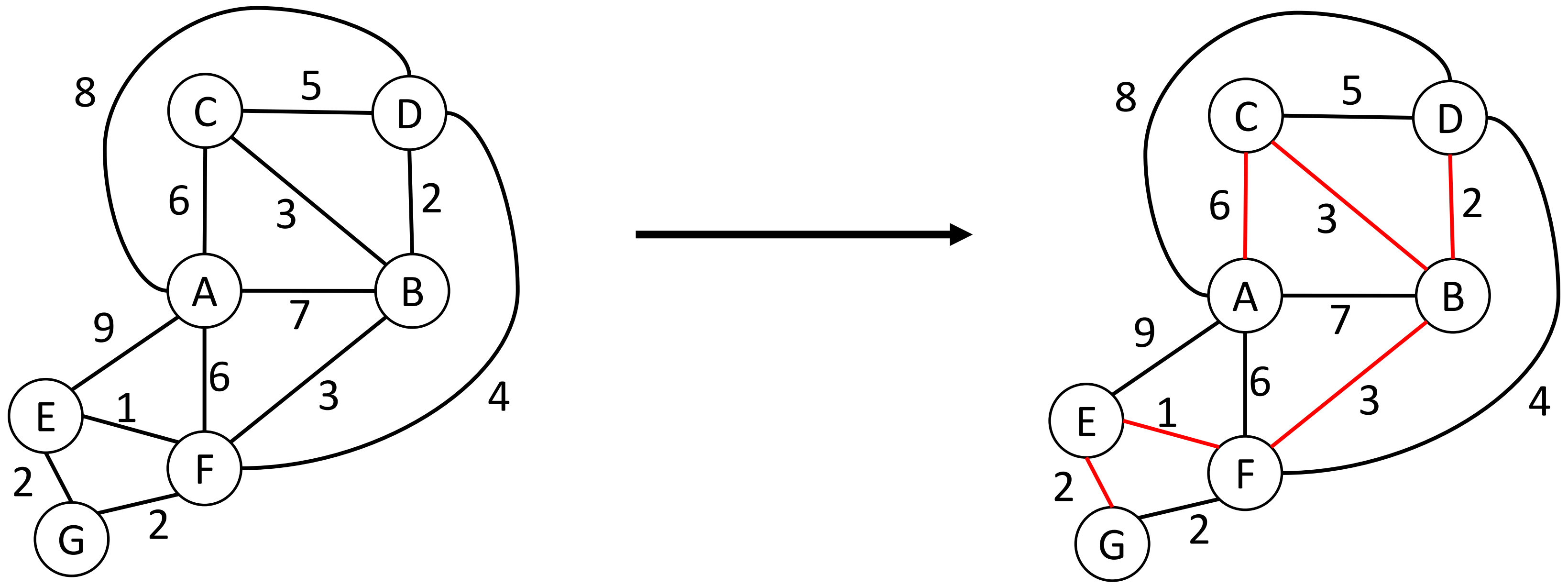
Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



Greedy MST

Compute MST on this undirected, weighted graph using a greedy approach:



Algorithmic paradigms: Summary

Generic framework that underlies a class of algorithms:

- Exhaustive search,
- Backtracking,

- Greedy algorithms
- Recursion
- Divide and conquer
- Dynamic programming

- Sweep-line algorithms

Exhaustive (or brute force) Search

Example problems:

1. Sorting an integer array of n values

2. (Decoding) Given a binary string m of length n, find the closest binary string to s within some subset C of binary strings of length n.
   - Example: $C = \{0000, 0101, 1010, 1111\}$ and $s = 1110$
   - HammingDistance(s_1, s_2) = number of positions in which the bits of s_1 and s_2 differ
   - Find string $s' \in C$ whose distance to s is the smallest among strings in C

Exhaustive Search: Sorting

Sorting an integer array of n values

- There are $\mathrm{O}(n!)$ possible arrays (or candidate solutions):
  - $n! = n \ * \ (n-1) * \cdots * 1$
  - When generating an array (or candidate solution), there are n values you can put in the first position, $n-1$ values in the second, and so on…
- Algorithm = For each generated array, check if it is sorted
  - Checking if it is sorted takes $O(n)$ time
  - At most $\mathrm{O}(n!)$ arrays are checked
  - In total, $O(n! * n)$ time

Exhaustive Search: Sorting

(Decoding) Given a binary string m of length n, find the closest binary string to s within some subset C of binary strings of length n.

- There are $\mathrm{O}(|C|) = O(2^n)$ possible bit strings (or candidate solutions)
  - (When generating a bit string (or candidate solution), each bit can be set to either 0 or 1)

- Algorithm = For each generated bit string s_i, compute its distance to s and keep s_i if it is the closest bit string found until now
  - Computing distance takes $O(n)$ time
  - At most $\mathrm{O}(2^n)$ bit strings are checked
  - In total, $O(2^n * n)$ time