COSC 2430

(a) What is the difference between shallow and deep equality tests on Arrays in Java?

```
 int arr1 [] =  {10, 20};
int arr2 [] = arr.clone();
System.out.println(arr1==arr2);
```

Shallow compare:

```
System.out.println(Arrays.equals(arr1, arr2));
```

False

arr1 and arr2 are two references to two different objects. So when we compare arr1 and arr2, two reference variables are compared

True

(a) What is the difference between shallow and deep equality tests on Arrays in Java?

```
int arr1 [] =  {10, 20};
int arr2 [] = {10, 20};
```

Shallow compare:

```
System.out.println(Arrays.equals(arr1, arr2));
```

-> True

Arrays.equals() works fine and compares arrays contents.

What if the arrays contain arrays inside them or some other references which refer to different object but have same values.

(a) What is the difference between shallow and deep equality tests on Arrays in Java?

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        // inarr1 and inarr2 have same values
        int inarr1[] = {1, 2, 3};
        int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one
element
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Not Same

So *Arrays.equals()* is not able to do deep comparison.

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        // inarr1 and inarr2 have same values
        int inarr1[] = {1, 2, 3};
        int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one
element
        if (Arrays.deepEquals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Same

So *Arrays.deepEquals()* is able to do deep comparison.

Sample Exam question: Write a method/function that does a "deep" linked list copy.

What is the time and space complexity for copying a linked list with n items?

Will the complexity vary for singly/doubly linked list? Why or why not?

(b) How would you backup (copy) all elements of an array to a new array using a *single* Java statement?

```
String [] arr = new String[3];
for (int i=0; i<3; i++)
    arr[i] ="hello";
String [] arr2 = arr.clone();
for (int i=0; i<3; i++)
    System.out.println(arr2[i]);
```

hello

hello

hello

(C) Provide an algorithm/pseudocode for finding the penultimate (second-last) node in a doubly linked list where the last node is indicated by a null next reference.

```
Node<E> first = head;
        if (first==null || first.next==null)
            return null;
Node<E> second = fisrt.next;
while (second.next !=null)       {
            first = second;
           second = second.next;       }
return first;
```

Sample Exam question: Write a method/function that finds the middle element of a singly linked list?
What is the time and space complexity?
How would you reverse a singly linked list in exactly one pass , O(n) – Hint: use a stack.

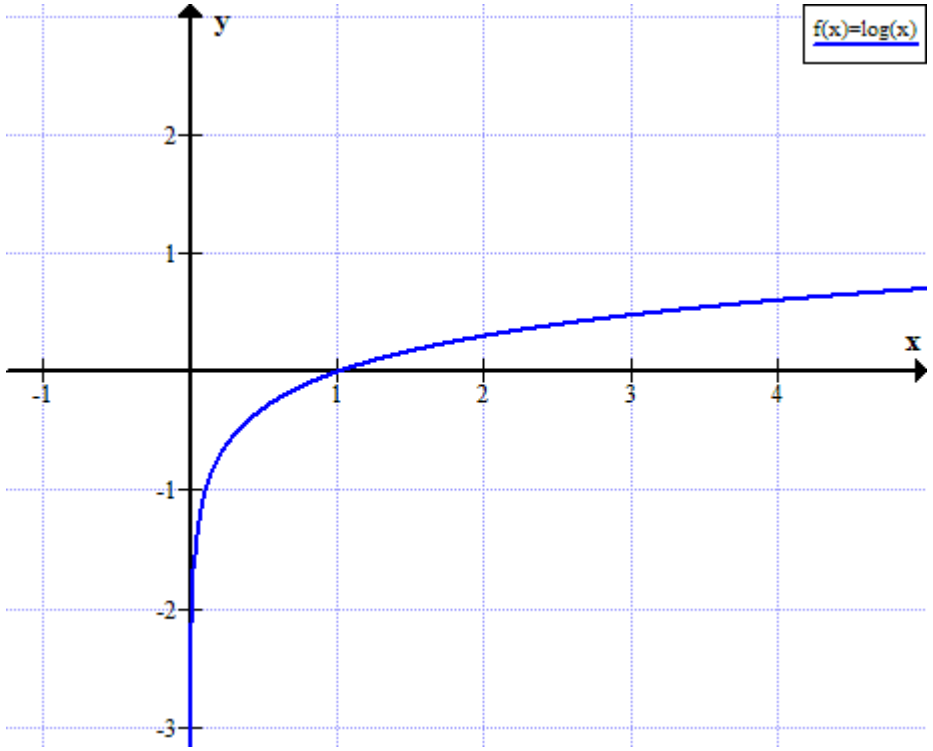
(d) Provide an algorithm/pseudocode to find the kth last element of a singly linked list starting with only the header sentinel?

```
Node<E> fast = head;
i=1;
 while(fast != null && i<k)       {
          fast=fast.next;
          i+=1;        }
if (fast==null)
return null;
else
Return fast;
```

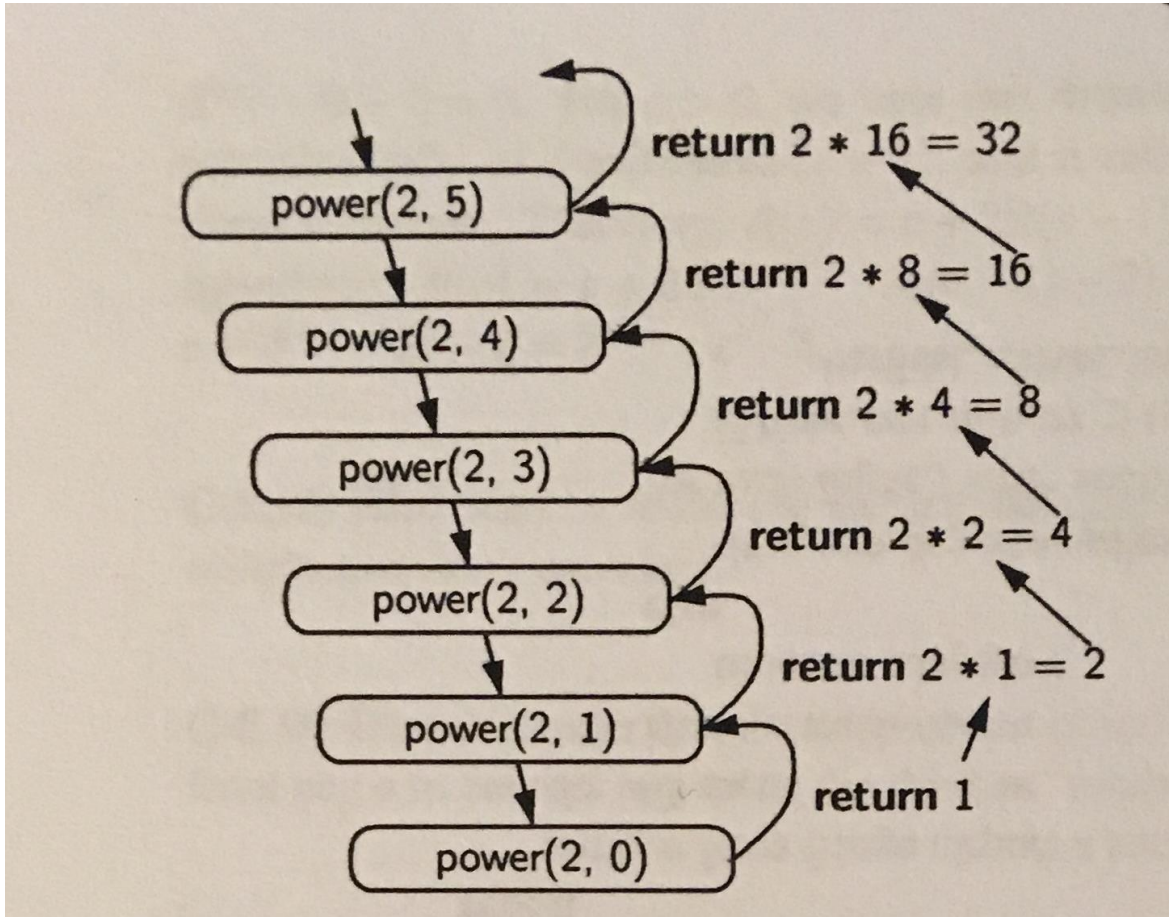
Sample Exam question: Write a method/function that finds the kth smallest element in a singly linked list.
What is the time and space complexity for the same?
Will the complexity vary for singly/doubly linked list? Why or why not?

(e) Which function has the similar profile (i.e., the same “shape”) in the log-log scale as it is in the classical y/x scale?

Y=log(x), an Increasing function



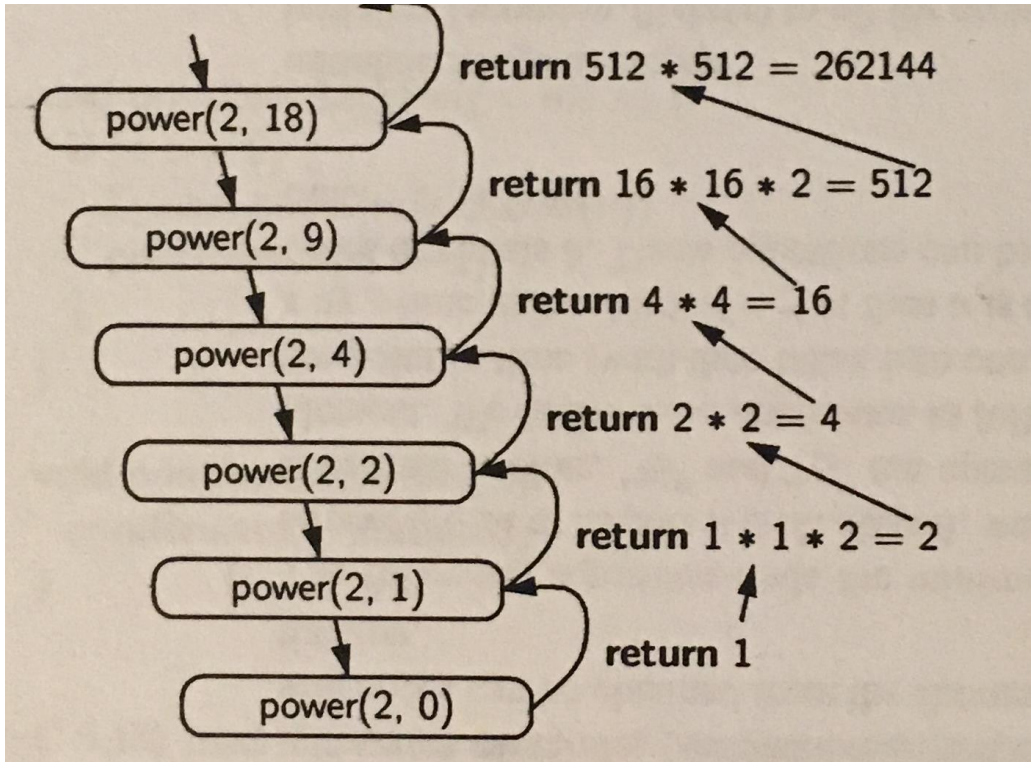
R-5.3 Draw the recursion trace for the computation of *power*(2,5), using the traditional algorithm implemented in Code Fragment 5.8.



R-5.4 Draw the recursion trace for the computation of *power*(2,18), using the repeated squaring algorithm, as implemented in Code Fragment 5.9.


```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else {
6      double partial = power(x, n/2);          // rely on truncated division of n
7      double result = partial * partial;
8      if (n % 2 == 1)                          // if n odd, include extra factor of x
9        result *= x;
10     return result;
11   }
12 }
```

Code Fragment 5.9: Computing the power function using repeated squaring.



R-5.9 Develop a nonrecursive implementation of the version of the power method from Code Fragment 5.9 that uses repeated squaring.

```
public static double power(double x, int n) {
  int k = 0;
  while ((1 << k) <= n)
    k++;

  double answer = 1;
  for (int j=k-1; j >= 0; j--) {
    answer *= answer;
    if (((1 << j) & n) > 0)
      answer *= x;
  }
  return answer;
}
```

<<:
Left shift
multiplying the number with some power of two.
1<<2 -> 4
1<<5 -> 32

C-5.20 Write a short recursive Java method that rearranges an array of integer values so that all the even values appear before all the odd values.

33

```
void organize(int[ ] data, int low, int high) {
  if (low < high) {
    if (data[high] & 1 == 0) {          // even
      int temp = data[high];
      data[high] = data[low];
      data[low] = temp;
      organize(data, low+1, high);      // data[low] is known to be even
    } else {
      organize(data, low, high−1);      // data[high] is known to be odd
    }
  }
}

void organize(int[ ] data) {
  organize(data, 0, data.length − 1);
}
```

R-6.3 What values are returned during the following series of stack operations, if executed upon an initially empty stack? push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

3, 8, 2,1,6, 7,4, 9

R-6.5 Give a recursive method for removing all the elements from a stack.

```
public void Remove(Stack S)
    {
 if (S.isEmpty())
        return;
      S.pop();
     Remove(S);     }
```

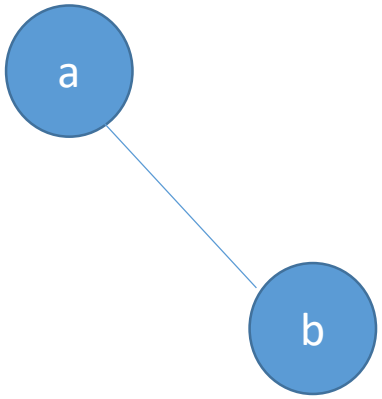
R-6.9 What values are returned during the following sequence of queue operations, if executed on an initially empty queue? enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().

- 5, 3, 2, 8, 9, 1, 7, 6

R-8.20 Let T be an ordered tree with more than one node. Is it possible that the preorder traversal of T visits the nodes in the same order as the postorder traversal of T? If so, give an example; otherwise, explain why this cannot occur. Likewise, is it possible that the preorder traversal of T visits the nodes in the reverse order of the postorder traversal of T? If so, give an example; otherwise, explain why this cannot occur.

- No, It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the same order.

- Yes, for example: pre-order: ab post-order: ba



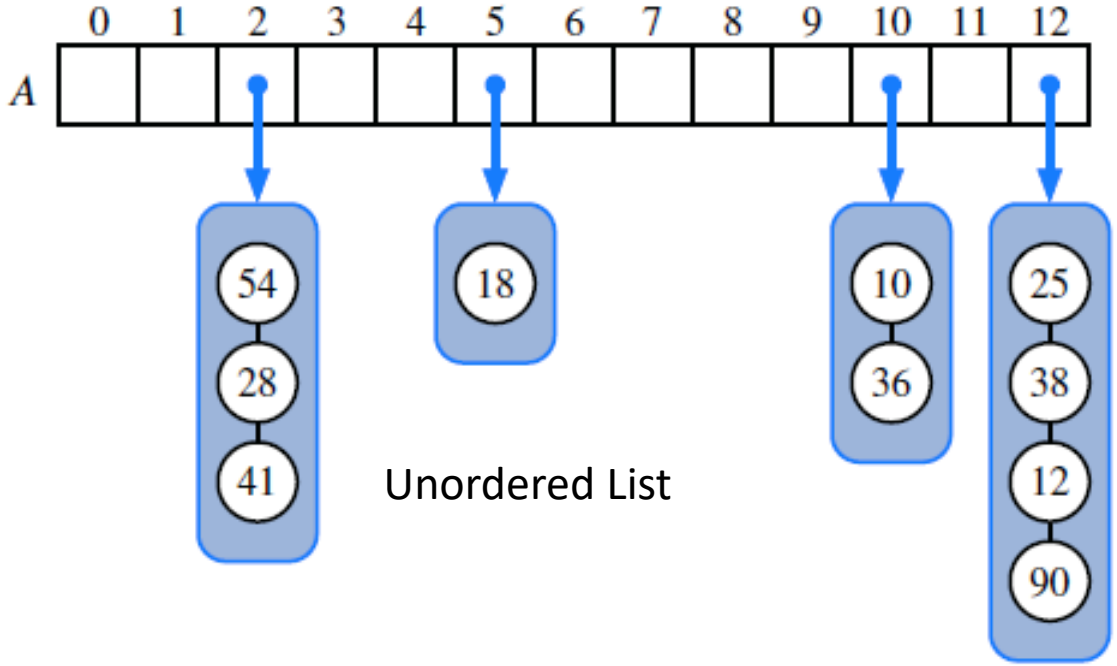
(k) When do collisions occur?
(l) What are two good collision handling shcmes?

Sol: When two keys have same hash values.

Sol: Seperate Chaning

Drawbacks:

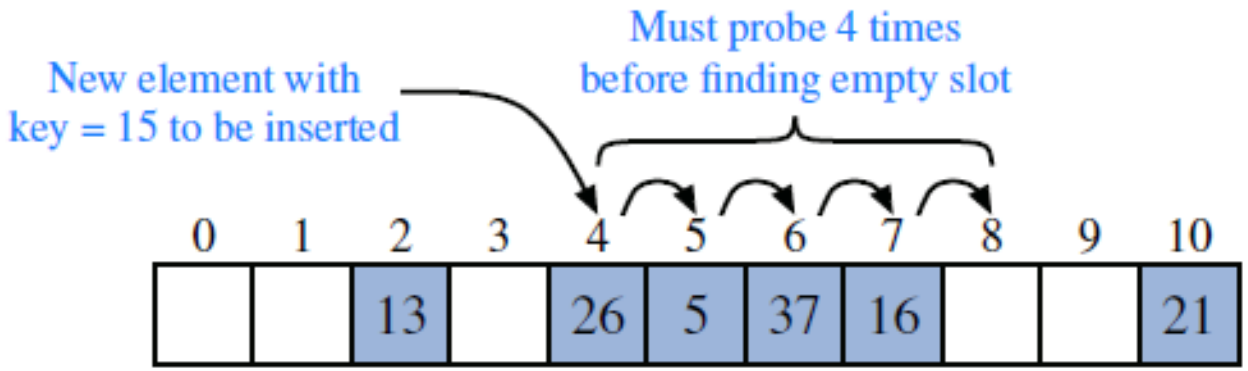
- It requires the use of an aux colliding keys



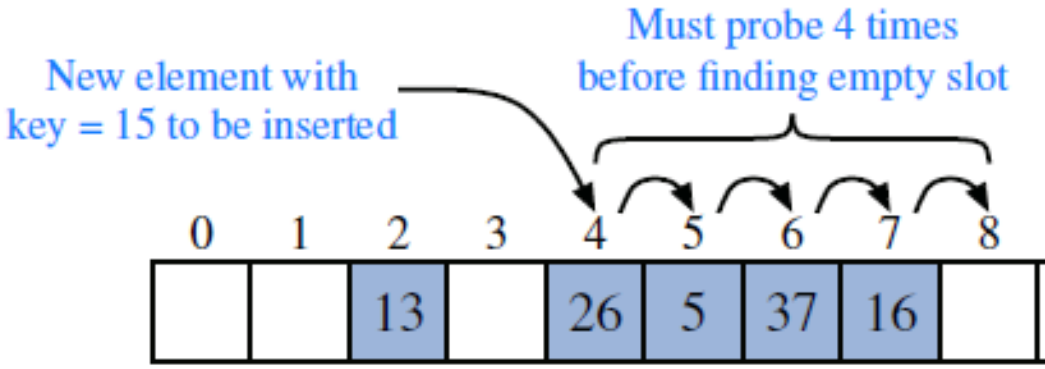
(k) When do collisions occur?
(l) What are two good collision handling shcmes?
When two keys have same hash values.

- Open Addressing

It requires that the ***load factor*** is always at most 1 and that entries are stored directly in the cells of the bucket array itself.



(k) When do collisions occur?
(l) What are two good collision handling shcmes?
When two keys have same hash values.



- Open Addressing (Linear Probing)

(k,v) j=h(k)

If A[j] is empty → insert (v) in A[j]

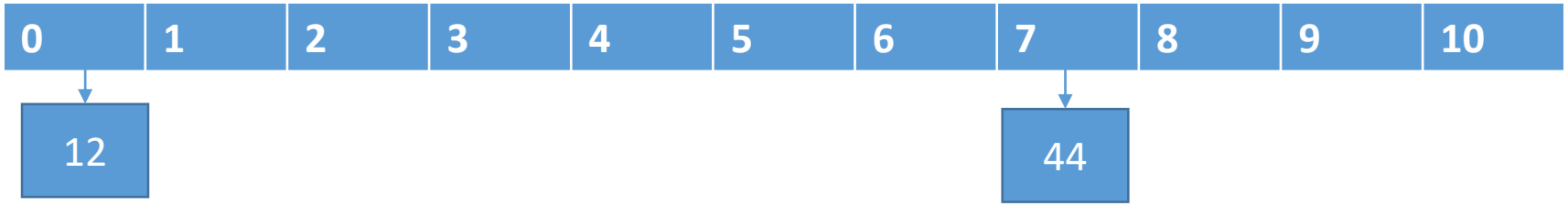
Else try -> insert (v) in A[(j+1) mod N]

If occupied try -> insert (v) in A[(j+2) mod N]

Draw the 11-entry hash table that results from using the hash function, h(i) = (4i+7) mod 11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

h(12) = (4 * 12 +7) mod 11 = 0

H(44) = (4 * 44 + 7) mod 11 = 7



Priority Queue

A collection of prioritized elements that allows arbitrary element insertion, allows the removal of the element that has first priority.
When an element is added to a priority queue,
the user designates its priority by providing an associated
key.
The element with the *minimal*
key will be the next to be removed
from
the queue

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Priority Queue- Unsorted list

- Doubly Link list is used to store <k,v> in PQ.
- Insert -> at the end of the list
- removeMin -> search for all items in list to find the minimum key

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Priority Queue- Sorted list

- Doubly Link list is used to store <k,v> in PQ.
- Items are sorted in the list in non-decreasing order
- The first item has maximum priority
- Insert -> start from end, scan backward
- removeMin -> remove first item

<1,C>,<2,A>,<4,B>

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

R-9.4 An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a *time stamp* that denotes the time when the event will occur. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time stamp (that is, add a future event).
- Extract the event with smallest time stamp (that is, determine the next event to process).

Which data structure should be used for the above operations? Why?

Priority Queue

Order : We need to retrieve the event with the smallest time stamp

Time stamp can be consider as the key and flight no as value

R-9.7 Illustrate the execution of the selection-sort algorithm on the following input sequence: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25).

22	15	36	44	10	3	9	13	29	25
3	15	36	44	10	22	9	13	29	25
3	9	36	44	10	22	15	13	29	25
3	9	10	44	36	22	15	13	29	25
3	9	10	13	36	22	15	44	29	25
3	9	10	13	15	22	36	44	29	25
3	9	10	13	15	22	36	44	29	25
3	9	10	13	15	22	25	44	29	36
3	9	10	13	15	22	25	29	44	36
3	9	10	13	15	22	25	29	36	44

Priority Queue: Selection Sort

- Selection Sort: at each step we select the next minimum.
- Similar to Priority Queue – Unsorted list
- Phase I : Insert all items from S to PQ
- Phase II: removeMin from PQ and add to the end of S.

		Sequence S	*Priority Queue P*
Input		(7, 4, 8, 2, 5, 3, 9)	()
Phase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(7, 4)
	⋮	⋮	⋮
	(g)	()	(7, 4, 8, 2, 5, 3, 9)
Phase 2	(a)	(2)	(7, 4, 8, 5, 3, 9)
	(b)	(2, 3)	(7, 4, 8, 5, 9)
	(c)	(2, 3, 4)	(7, 8, 5, 9)
	(d)	(2, 3, 4, 5)	(7, 8, 9)
	(e)	(2, 3, 4, 5, 7)	(8, 9)
	(f)	(2, 3, 4, 5, 7, 8)	(9)
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

Priority Queue: Insertion Sort

- Insertion Sort: at each step we Insert the item in its appropriate place.
- (keep the sub list sorted)
- Similar to Priority Queue –sorted list

		Sequence S	*Priority Queue P*
Input		(7, 4, 8, 2, 5, 3, 9)	()
Phase 1	(a)	(4, 8, 2, 5, 3, 9)	(7)
	(b)	(8, 2, 5, 3, 9)	(4, 7)
	(c)	(2, 5, 3, 9)	(4, 7, 8)
	(d)	(5, 3, 9)	(2, 4, 7, 8)
	(e)	(3, 9)	(2, 4, 5, 7, 8)
	(f)	(9)	(2, 3, 4, 5, 7, 8)
	(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2	(a)	(2)	(3, 4, 5, 7, 8, 9)
	(b)	(2, 3)	(4, 5, 7, 8, 9)
	⋮	⋮	⋮
	(g)	(2, 3, 4, 5, 7, 8, 9)	()

R-9.9 Give an example of a worst-case sequence with n elements for insertion-sort, and show that insertion-sort runs in $\Omega(n^2)$ time on such a sequence.

Input sequence is in decreasing order

44, 30, 22, 15, 12, 10, 3

44

30,44

22,30,44

15,22,30,44

10,15,22,30,44

3, 10,15,22,30,44

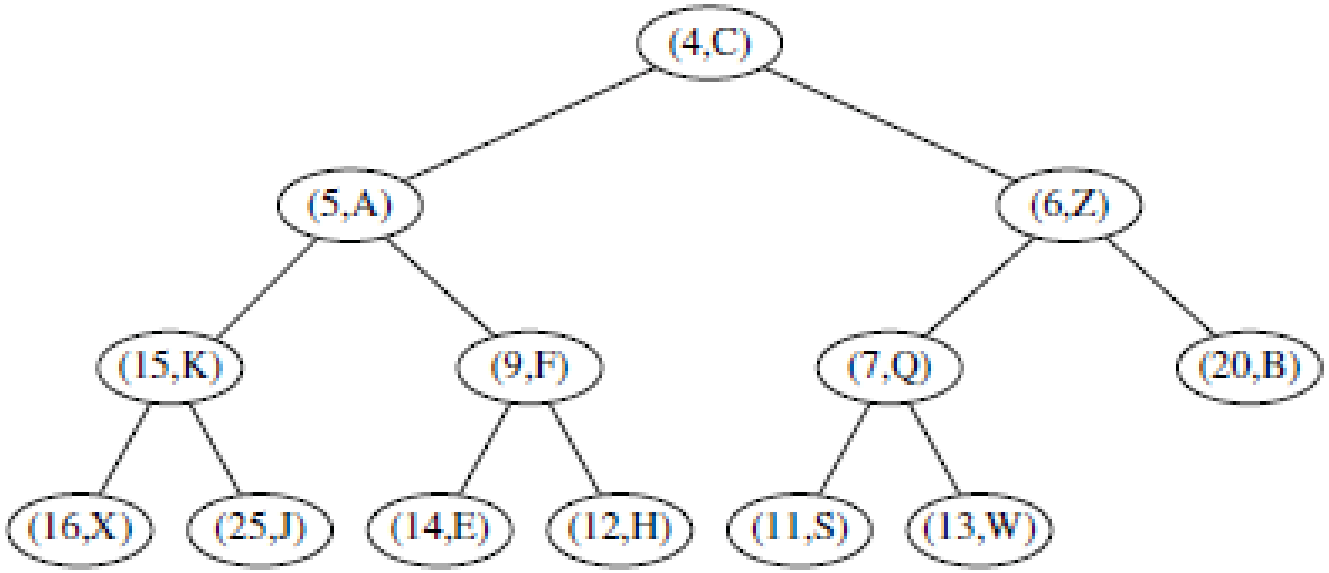
For each Item we start from end, moves backward and Insert at the beginning

O(n^2)

Heap

- ***Heap-Order Property*:** In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.
- ***Complete Binary Tree Property*:** A heap T with height h is a ***complete*** binary tree if levels 0,1,2, . . . ,$h-1$ of T have the maximal number of nodes possible (namely, level i has $2i$ nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level.

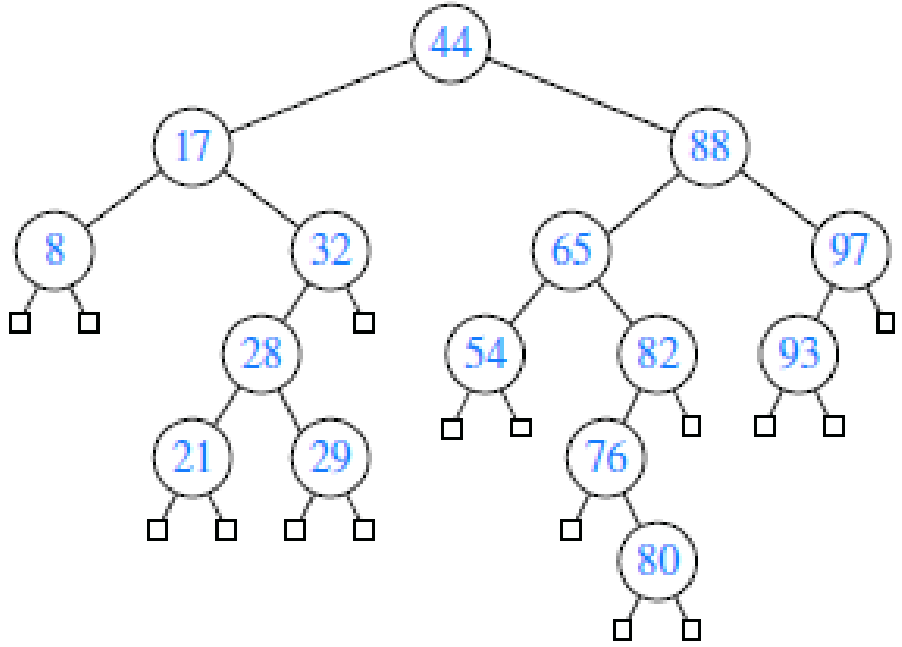
R-9.11 At which positions of a heap might the largest key be stored?



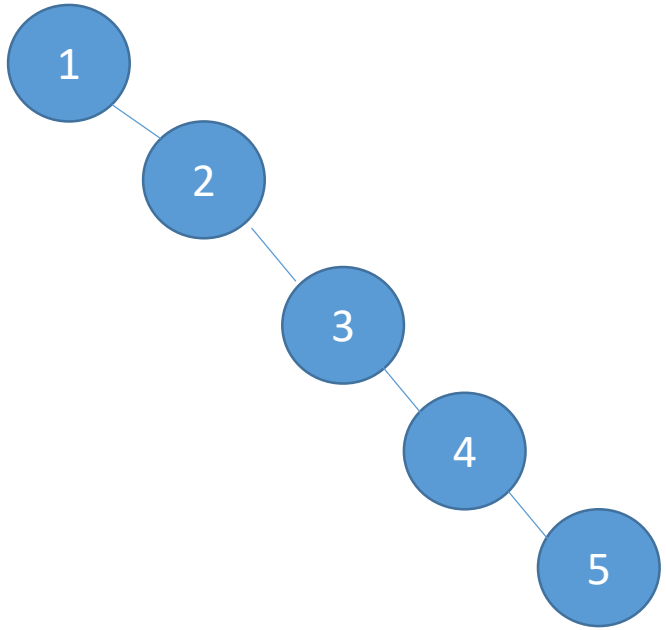
- If the smallest is at the top of the heap, the largest key in a heap may be at any external node (child, leaf)

BST

- each internal position *p* stores a key-value pair (*k*,*v*) such that:
- Keys stored in the left subtree of *p* are less than *k*.
- Keys stored in the right subtree of *p* are greater than *k*.



R-11.1 If we insert the entries $(1,A)$, $(2,B)$, $(3,C)$, $(4,D)$, and $(5,E)$, in this order, into an initially empty binary search tree, what will it look like?



R-12.20 Suppose S is a sequence of n values, each equal to 0 or 1. How long will it take to sort S with the merge-sort algorithm? What about quick-sort?

- Merge Sort O(nlog n)
- Quick Sort O(n)
- At first we choose a pivot then rearrange the items such that :
- All item on the left side of the pivot <= pivot
- All items on the right side of the pivot >= pivot

=>> O(n) => done!

R-12.23 Give an example input that requires merge-sort and heap-sort to take $O(n \log n)$ time to sort, but insertion-sort runs in $O(n)$ time. What if you reverse this list?

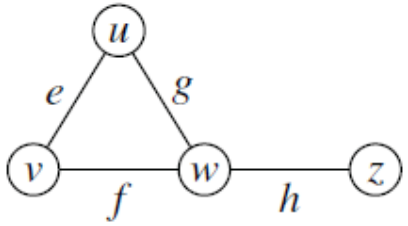
- An input list that is already sorted

If we reverse the input:

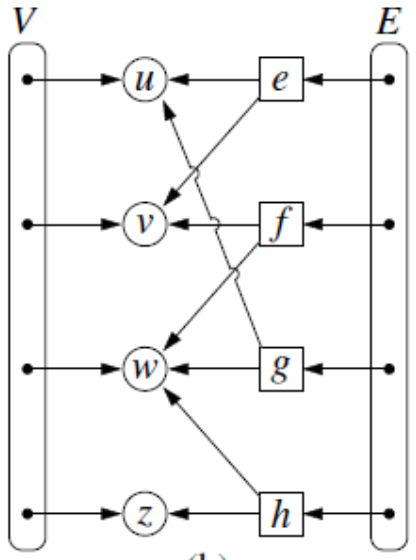
- Marge-sort , heap-sort -> O(n log n)
- Insertion-sort -> O(n^2)

Graph Representations

- In an ***edge list***, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (*u*,*v*), or the set of all edges incident to a vertex *v*.



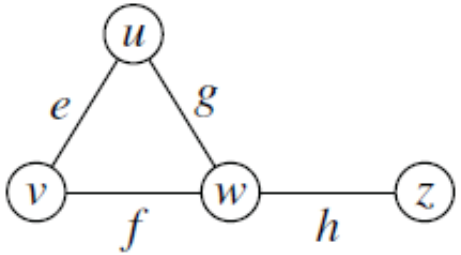
(a)



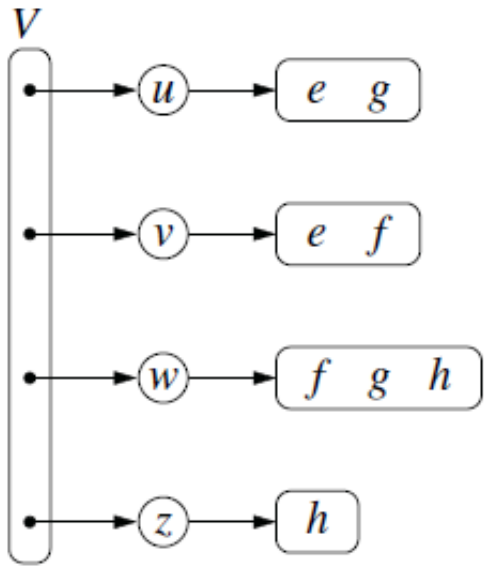
(b)

Graph Representations

- In an ***adjacency list***, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex.



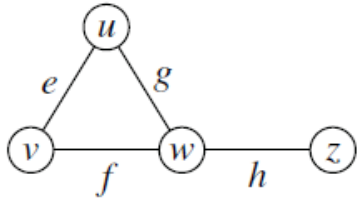
(a)



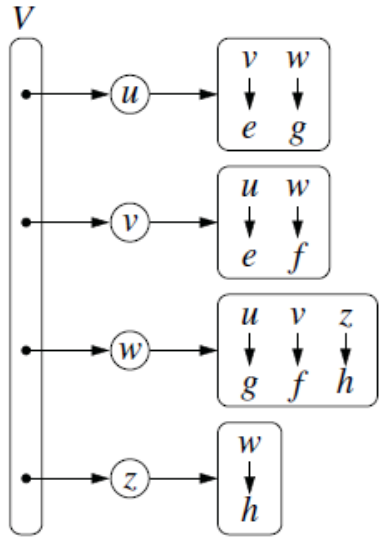
(b)

Graph Representations

- An ***adjacency map*** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (*u*,*v*), for example, in *O*(1) expected time with hashing.



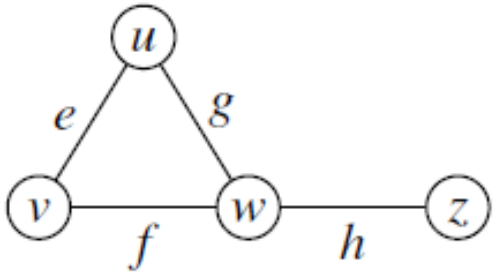
(a)



(b)

Graph Representations

- An ***adjacency matrix*** provides worst-case $O(1)$ access to a specific edge (u,v) by maintaining an $n \times n$ matrix, for a graph with n vertices. Each slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null.



(a)

		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

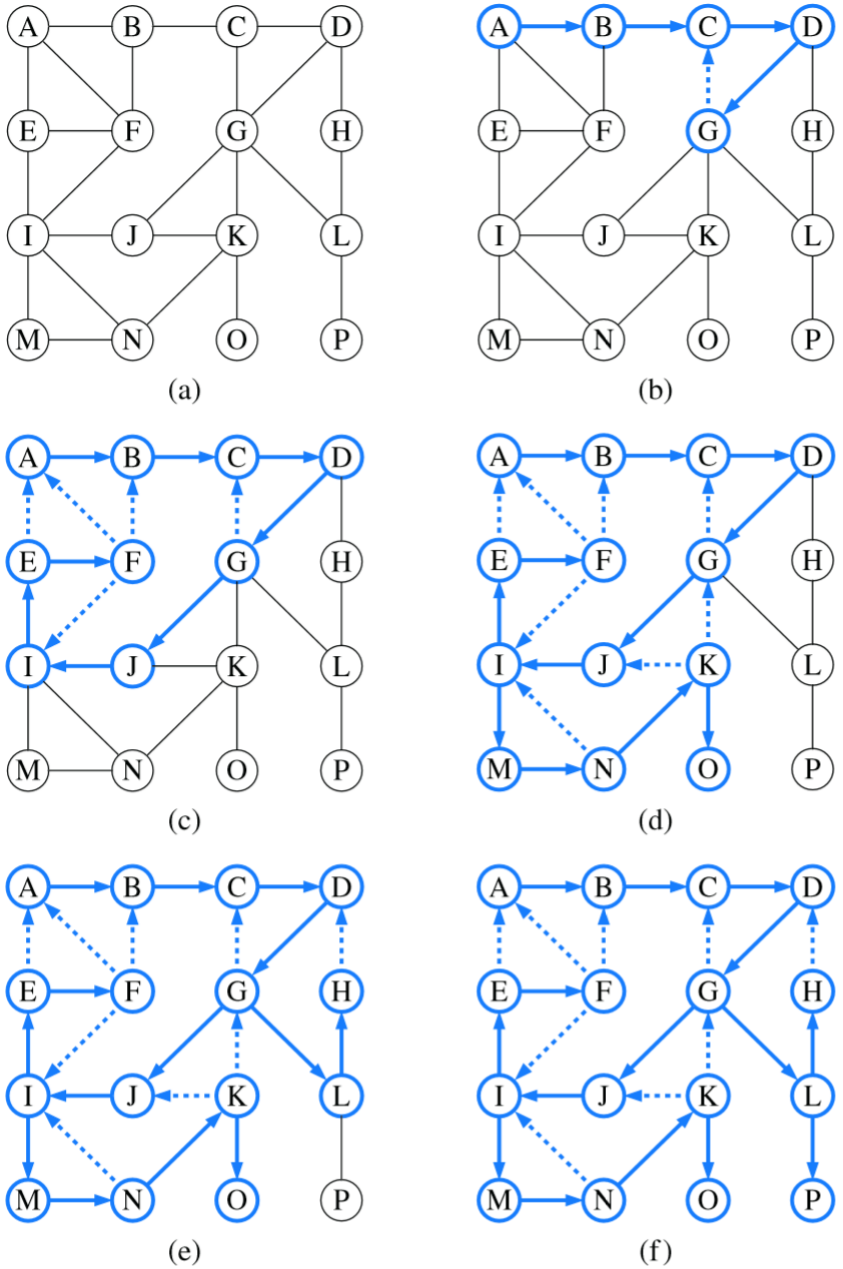
(b)

Graph Traversals

- Depth first search and breadth first search also work for arbitrary (directed or undirected) graphs
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine *connectivity*:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Important difference: Breadth-first search always finds a shortest path from the start vertex to any other (for unweighted graphs)
  - Depth first search may not!

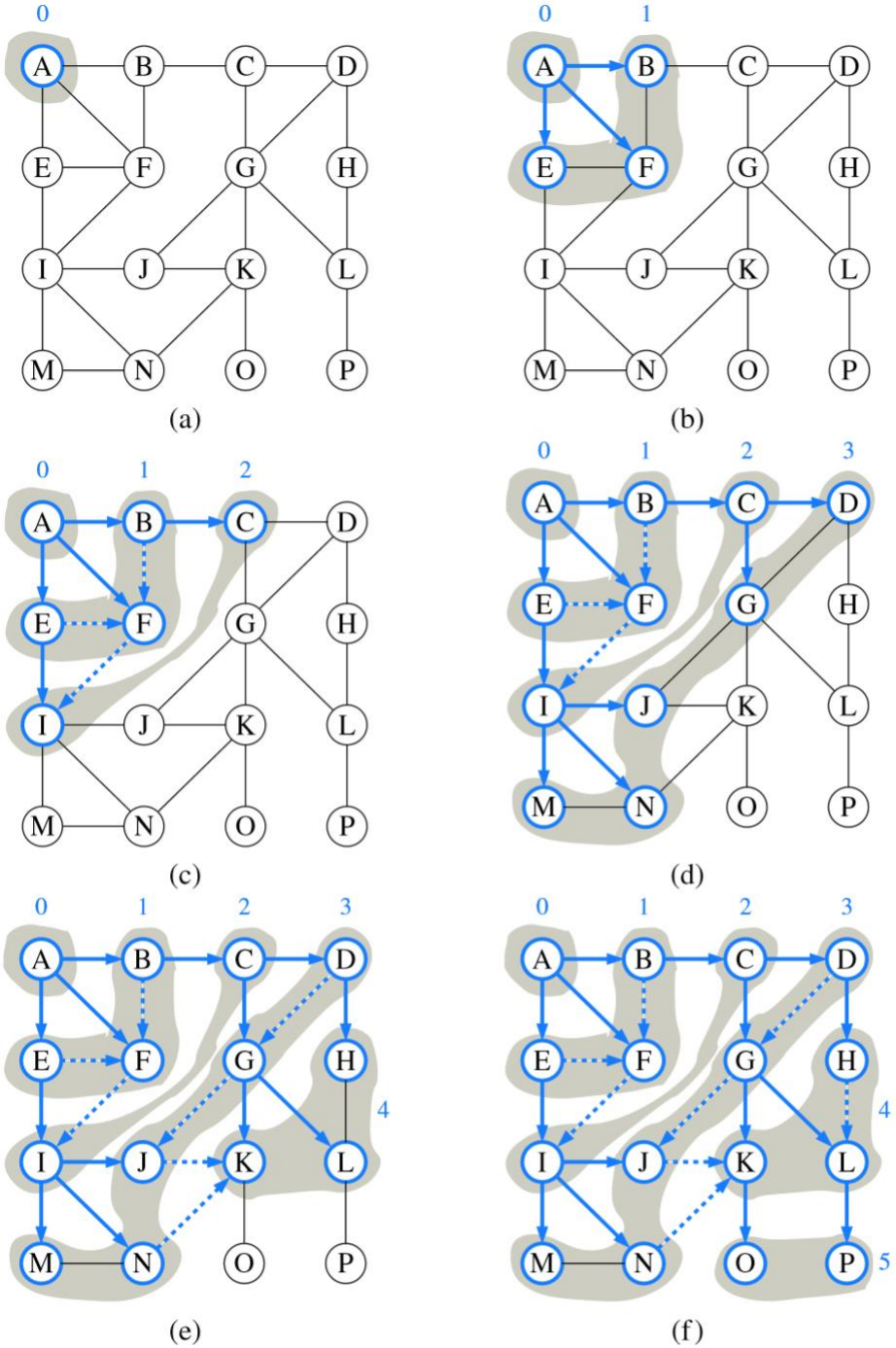
Depth First Search

Algorithm DFS(G, u):
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u, v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).



Breadth First Search

```
1  /** Performs breadth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
5    known.add(s);
6    level.addLast(s);                          // first level includes only s
7    while (!level.isEmpty()) {
8      PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
9      for (Vertex<V> u : level)
10       for (Edge<E> e : g.outgoingEdges(u)) {
11         Vertex<V> v = g.opposite(u, e);
12         if (!known.contains(v)) {
13           known.add(v);
14           forest.put(v, e);                    // e is the tree edge that discovered v
15           nextLevel.addLast(v);                // v will be further considered in next pass
16         }
17       }
18     level = nextLevel;                         // relabel 'next' level to become the current
19   }
20 }
```



Is BFS the Hands Down Winner?

- Depth-first search
  - Simple to implement (implicit or explict stack)
  - Does not always find shortest paths
  - Must be careful to “mark” visited vertices, or you could go into an infinite loop if there is a cycle
- Breadth-first search
  - Simple to implement (queue)
  - Always finds shortest paths
  - Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate

Edsger Wybe Dijkstra
(1930-2002)

- Invented concepts of structured programming, synchronization, weakest precondition, and "semaphores" for controlling computer processes. The Oxford English Dictionary cites his use of the words "vector" and "stack" in a computing context.
- Believed programming should be taught without computers
- 1972 Turing Award
- “In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”

Dijkstra's Algorithm for Single Source Shortest Path

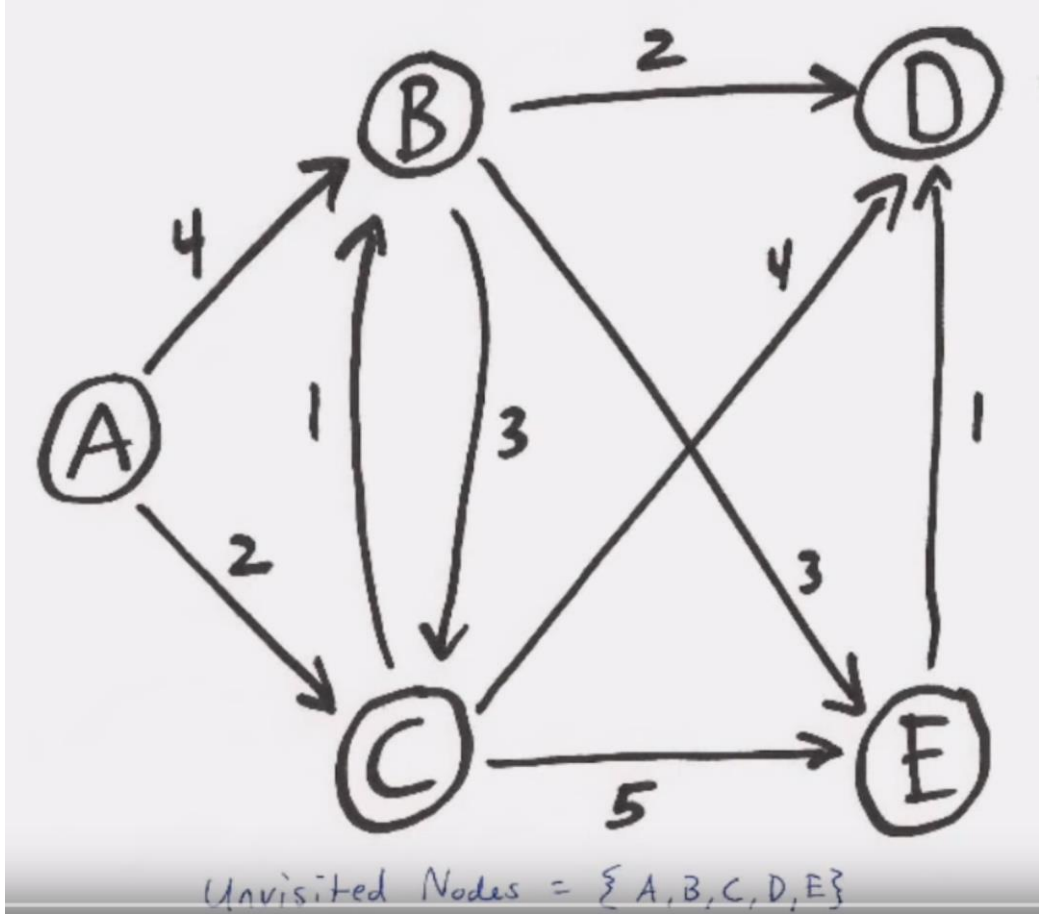
- Classic algorithm for solving shortest path in weighted graphs (with *only positive* edge weights)
- Similar to breadth-first search, but uses a priority queue instead of a FIFO queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
  - a kind of "greedy" algorithm
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

Pseudocode for Dijkstra

Algorithm ShortestPath(G, s):
Input: A directed or undirected graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
{pull a new vertex u into the cloud}
$u =$ value returned by Q.removeMin()
for each edge (u, v) such that v is in Q **do**
{perform the ***relaxation*** procedure on edge (u, v)}
if $D[u] + w(u, v) < D[v]$ **then**
$D[v] = D[u] + w(u, v)$
Change the key of vertex v in Q to $D[v]$.
return the label $D[v]$ of each vertex v

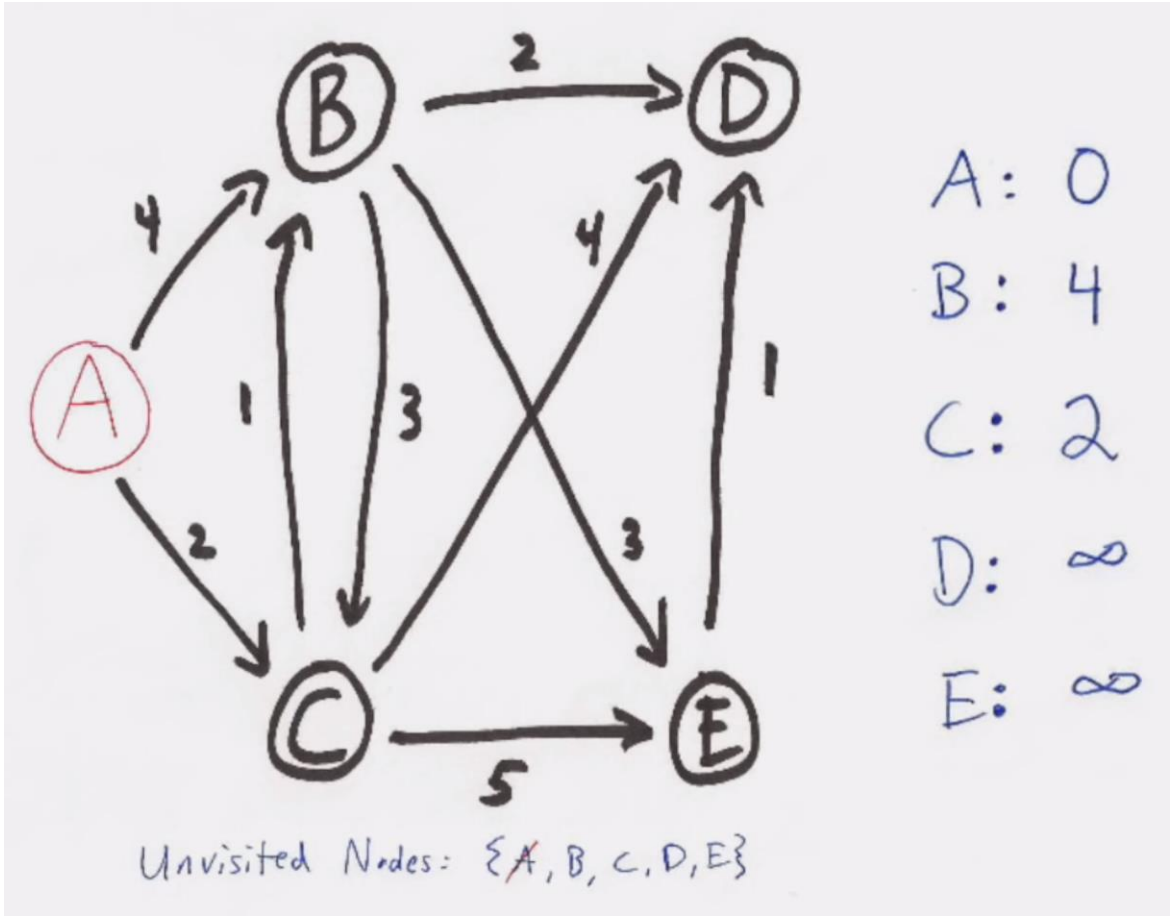
Code Fragment 14.12: Pseudocode for Dijkstra's algorithm, solving the single-source shortest-path problem for an undirected or directed graph.

Very Simple Example

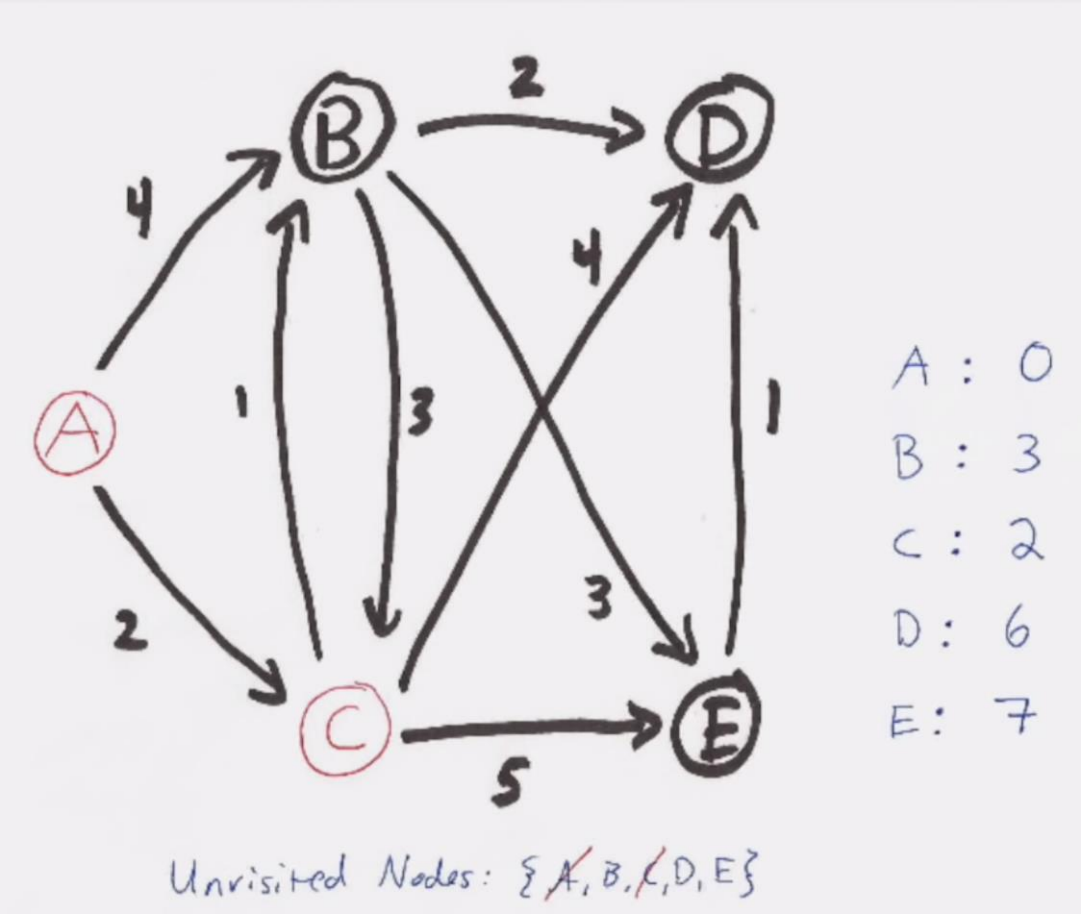


https://www.youtube.com/watch?v=_lHSawdgXpI

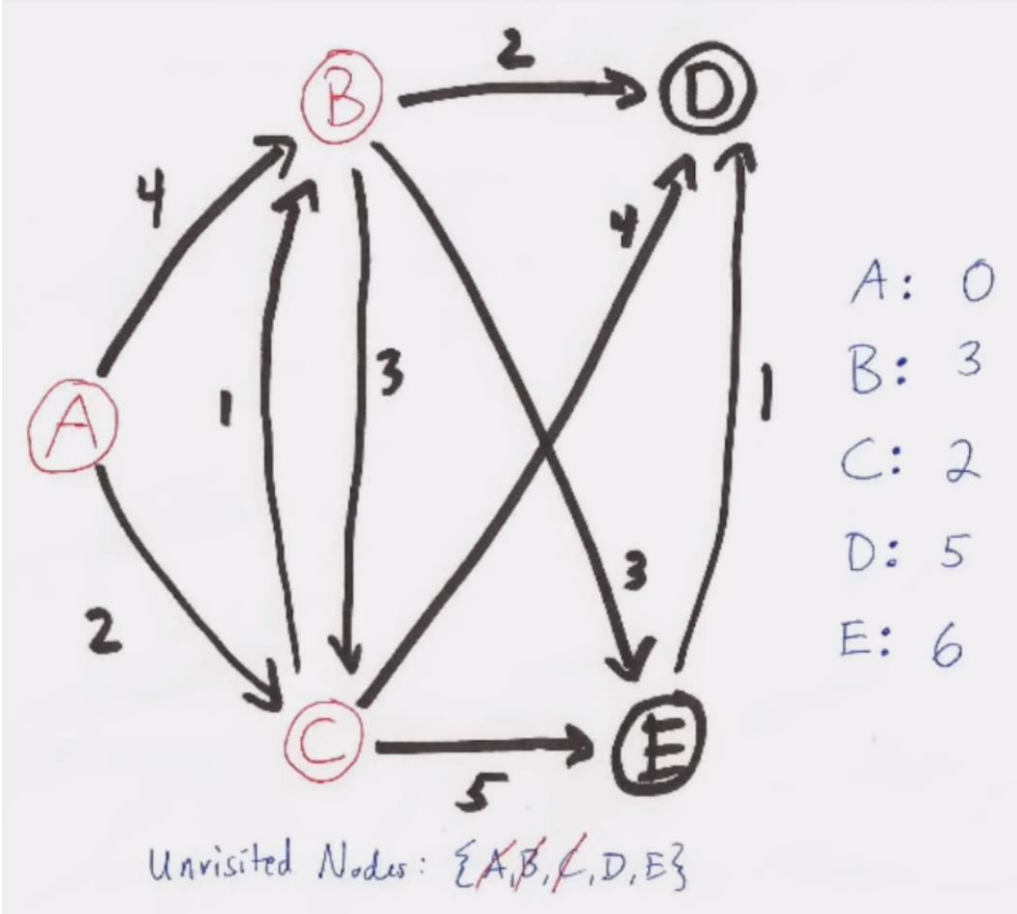
Very Simple Example



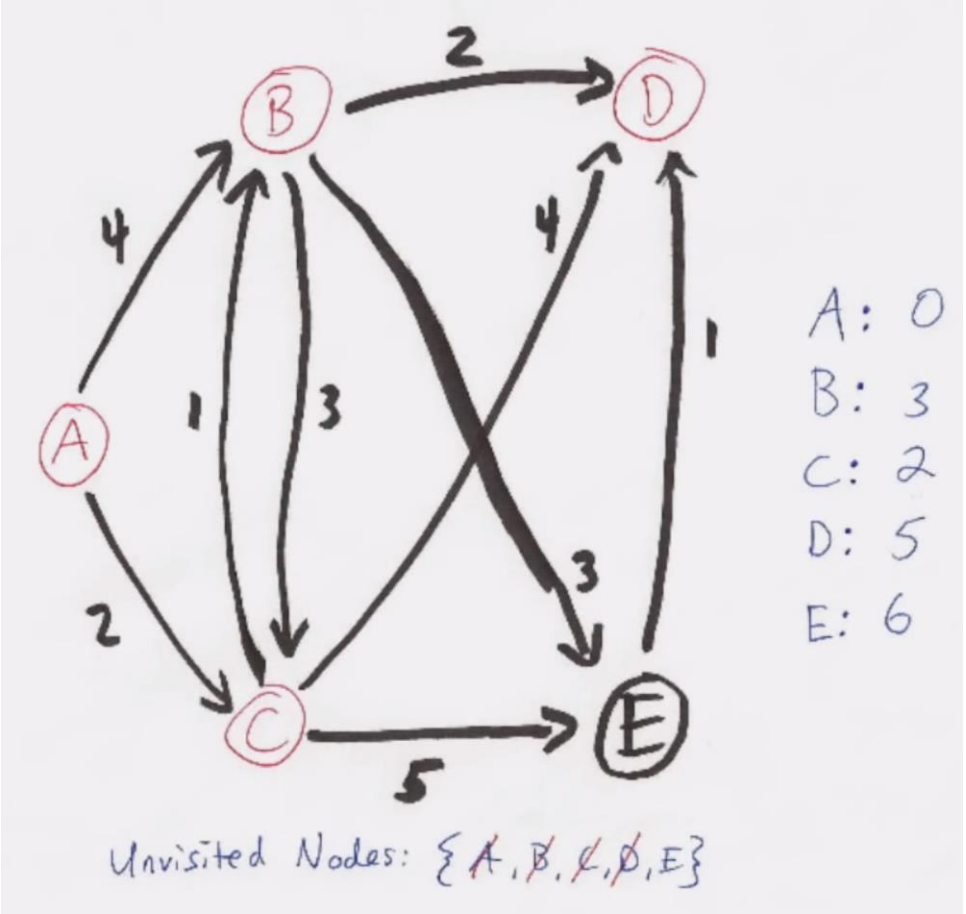
Very Simple Example



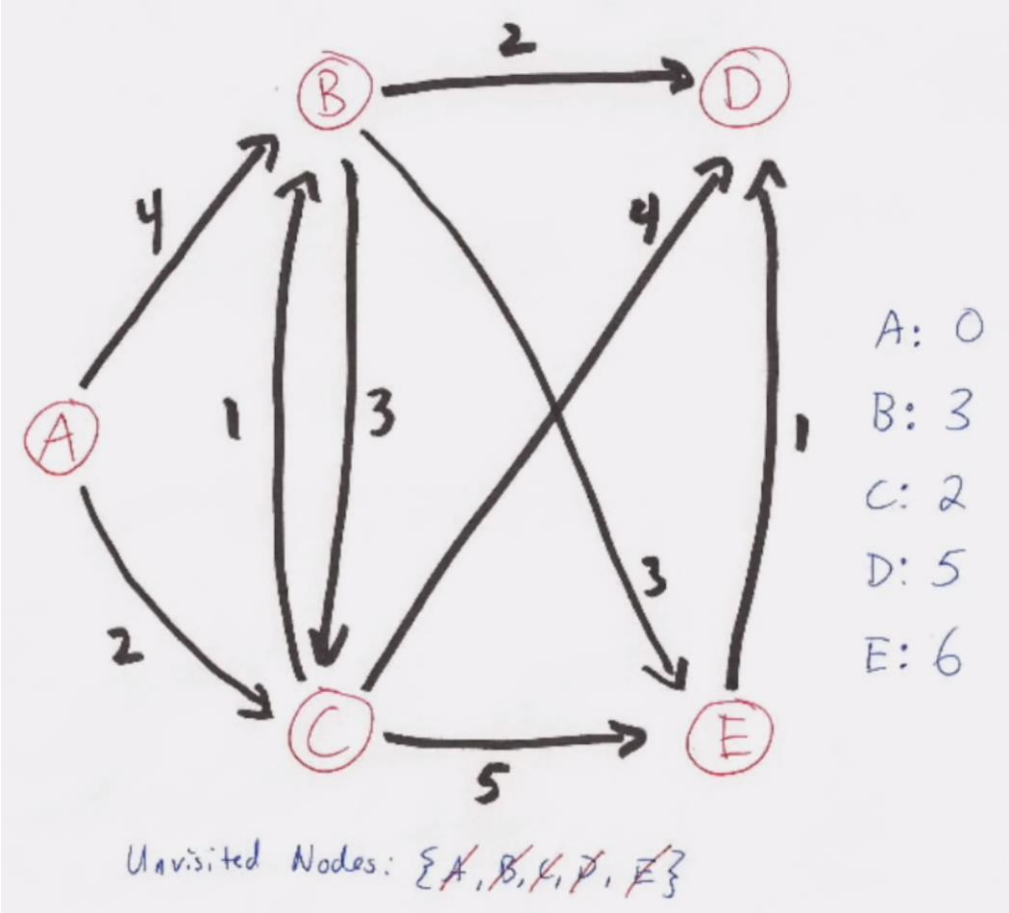
Very Simple Example



Very Simple Example



Very Simple Example



Very Simple Example