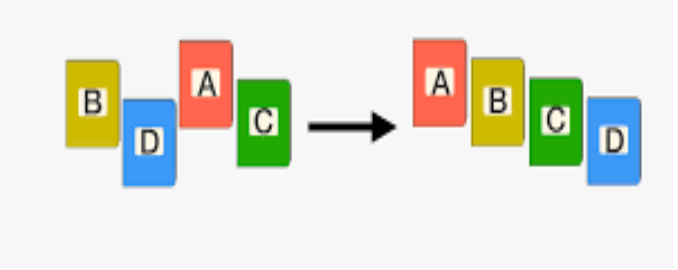
1. Introduction to Algorithms and Review of Data Structures

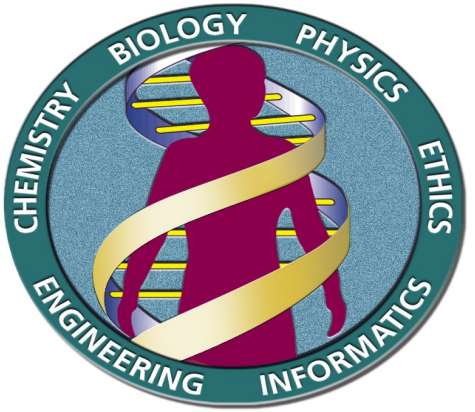
Pukar Karki
Assistant Professor

Algorithm

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output.

What kind of problems are solved by algorithms?





FedEx

DuckDuckGo

Properties of an Algorithm

1. Input specified
2. Output specified
3. Definiteness
4. Effectiveness
5. Finiteness
6. Independent

Expressing Algorithms

An algorithm may be expressed in a number of ways:

- **natural language**
  - – usually verbose and ambiguous;
- **flow charts**
  - – avoids most (if not all) issues of ambiguity;
  - – difficult to modify w/o specialized tools;
  - – largely standardized
- **pseudo-code**
  - – also avoids most issues of ambiguity;
  - – vaguely resembles common elements of programming languages;
  - – no particular agreement on syntax
- **programming language**
  - – tend to require expressing low-level details that are not necessary for a high-level understanding

The Random Access Machine (RAM) Model

- A machine has a **CPU** and a **memory**.
- **Memory** :
  - An infinite sequence of cells, each of which contains the same number **w** of bits.
  - Every cell has an address: the first cell of memory has address 1,the second cell 2, and so on.

The Random Access Machine (RAM) Model

- **CPU**

  – Contains a fixed numbers of registers,each of which has **w** bits (i.e., same as a memory cell).

The Random Access Machine (RAM) Model

- **CPU**

  – Can do the following 4 atomic operations

  1.Register (Re-)Initialization : Set a register to a fixed value (e.g., 0,−1, 100, etc.), or to the contents of another register.

The Random Access Machine (RAM) Model

- **CPU**

  – Can do the following 4 atomic operations

  2.Arithmetic : Take the integers a,b stored in two registers, calculate one of the following and store the result in a register:

  a+b, a−b, a*b, and a/b.

The Random Access Machine (RAM) Model

- **CPU**
  - – Can do the following 4 atomic operations

  3.Comparison/Branching : Take the integers a,b stored in two registers, compare them,and learn which of the following is true:

  a<b ,a=b, a>b.

The Random Access Machine (RAM) Model

- **CPU**
  - – Can do the following 4 atomic operations

  4.Memory Access : Take a memory address A currently stored in a register. Do one of the following:

    – Read the contents of the memory cell with address A into another register (overwriting the bits there).

    – Write the contents of another register into the memory cell with address A(overwriting the bits there).

The Random Access Machine (RAM) Model

- **Algorithms and Their Cost**

  – An algorithm is a sequence of atomic operations.

  – Its cost(also called its running time, or simply,time) is the length of the sequence, namely, the number of atomic operations.

Example

Problem

Suppose that an integer of $n \geq 1$ has already been stored at the memory cell of address 1. We want to calculate $1 + 2 + \ldots + n$(the sum can be stored anywhere, e.g., in a register).

Example

Algorithm #1

1.load n into register a

2.register b ← 0, c ← 1, d ← 1.

3.repeat

4.b ← b+c

5.c ← c+d

6.until c>a

7.return b

Cost of the algorithm = 3n+ 4 = O(n)

Example

Algorithm #2

1.register a ← 1, b ← 2, c ← n.

2.Set a ← a+c *(note:a now equals n+ 1).*

3. Set a ← a∗c *(now a equals n(n+ 1)).*

4. Set a ← a/b *(now a equals n(n+ 1)/2).*

*5. R*eturn a;

Cost of the algorithm = 6 = O(1)

Time and Space Complexity

- **Time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Similarly, **Space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and Space Complexity

- Time and space complexity depends on lots of things like hardware, operating system, processors, etc.
- However, we don't consider any of these factors while analyzing the algorithm.
- We will make use of the RAM model to compute the number of atomic operations(time complexity) and the number of memory cells it uses(space complexity).

Best, Worst and Average Case

- Best case complexity gives **lower bound** on the running time of the algorithm for any instance of input(s).
- This indicates that the algorithm can never have lower running time than best case for particular class of problems.

Best, Worst and Average Case

- Worst case complexity gives **upper bound** on the running time of the algorithm for all the instances of the input(s).
- This ensures that no input can overcome the running time limit posed by worst case complexity.

Best, Worst and Average Case

- Average case complexity gives **average number of steps** required on any instance(s) of the input(s).

Detailed Analysis of an Algorithm

Example 1 : algorithm to compute the factorial

```
fact(n)
{
        f = 1;
        for(i = 1; i<=n; i++)
            {
            f = f * i;
            }
        return f;
}
```

Detailed Analysis of an Algorithm

Example 1 : algorithm to compute the factorial

```
fact(n)
{
        f = 1;
        for(i = 1; i<=n; i++)
            {
            f = f * i;
            }
        return f;
}
```

f = 1 takes 1 unit

Inside for loop

- i = 1 takes 1 unit
- i <= n takes n+1 unit
- i++ takes n unit
- f = f * i takes n unit

return f takes 1 unit

Time Complexity = 1 + 1 + (n+1) + n + n + 1 = 3n + 4 = O(n)

Detailed Analysis of an Algorithm

Example 1 : algorithm to compute the factorial

```
fact(n)
{
        f = 1;
        for(i = 1; i<=n; i++)
            {
            f = f * i;
            }
        return f;
}
```

- f takes 1 unit of memory cell
- i takes 1 unit of memory cell
- n takes 1 unit of memory cell

Space Complexity = 1 + 1 + 1 = 3 = O(1)

Detailed Analysis of an Algorithm

Example 2 : bubble sort

Input: *an array of size **n***

Output: *the input array of size **n** in sorted form*

Algorithm:

```
for(int i =1;i<n;i++)
{
    for(int j=0;j<n-i;j++)
    {
        if (a[j] > a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }

    }
}
```

Detailed Analysis of an Algorithm

Example 2 : bubble sort

```
for(int i =1;i<n;i++)
{
    for(int j=0;j<n-i;j++)
    {
        if (a[j] > a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }

    }
}
```

Space Complexity

- i takes 1 unit of memory cell
- j takes 1 unit of memory cell
- temp takes 1 unit of memory cell
- n takes 1 unit of memory cell
- Array takes n unit of memory cell

=1+1+1+1+n

=n+4

=O(n)

Detailed Analysis of an Algorithm

Example 2 : bubble sort

```
for(int i =1;i<n;i++)
{
    for(int j=0;j<n-i;j++)
    {
        if (a[j] > a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }

    }
}
```

Time Complexity

Within first for loop:

- i=1 takes 1 step
- i<n takes n steps
- i++ takes (n-1) steps

Inside second for loop:

- j=0 takes (n-1) step
- j<n-i takes [n+(n-1)+(n-2)+ _ _ _ _ + 2 +1]
- j++ takes [(n-1) + (n-2) + _ _ _ _ _ +2 + 1]

Detailed Analysis of an Algorithm

Example 2 : bubble sort

```
for(int i =1;i<n;i++)
{
    for(int j=0;j<n-i;j++)
    {
        if (a[j] > a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }

    }
}
```

Time Complexity

In if statement: It takes at most 3*(n-1)

= **1+n+(n-1)**

+ *[n+{n+(n-1)+(n-2)+ _ _ _ _ +3+2+1]+[(n-1)+ (n-2)+_ _ _ _ +3+2+1}]*

+***3*[(n-1)+(n-2)+_ _ _ _ +3+2+1]***

= **2n**+*[n+n(n+1)/2+n(n-1)/2]*+***3*n(n-1)/2**

= **2n**+*n+(n^2)/2+n/2+(n^2)/2-n/2*+***(3n^2)/2-3n/2***

= ($5n^2$+3n)/2

= [O(1)*O(n^2)+O(1)*O(n)]/O(1)

= [O(n^2)+O(n)]/O(1)

= O(n^2)+O(n)

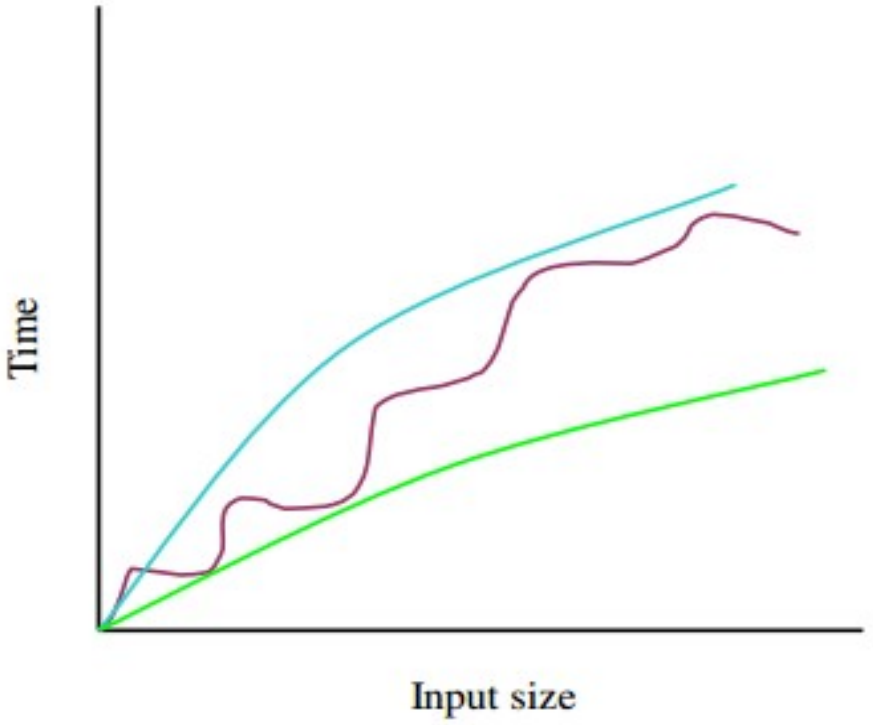
= O(n^2)

Asymptotic Notations

- Complexity analysis of an algorithm is very hard if we try to analyze exact.
- We know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input.
- So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier.

Asymptotic Notations

- For this purpose we need the concept of asymptotic notations.
- The figure below gives upper and lower bound concept.



Asymptotic Notations

- Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- This is also known as an algorithm's growth rate.

Asymptotic Notations

- Does the algorithm suddenly become incredibly slow when the input size grows?
- Does it mostly maintain its quick run time as the input size increases?
- Asymptotic Notation gives us the ability to answer these questions.

Asymptotic Notations

- These are some basic function growth classifications used in various notations.
- The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time.)

  $1 < \log(n) < SQRT(n) < n < n*\log(n) < n^2 < n^3 ... 2^n < 3^n ... < n^n$

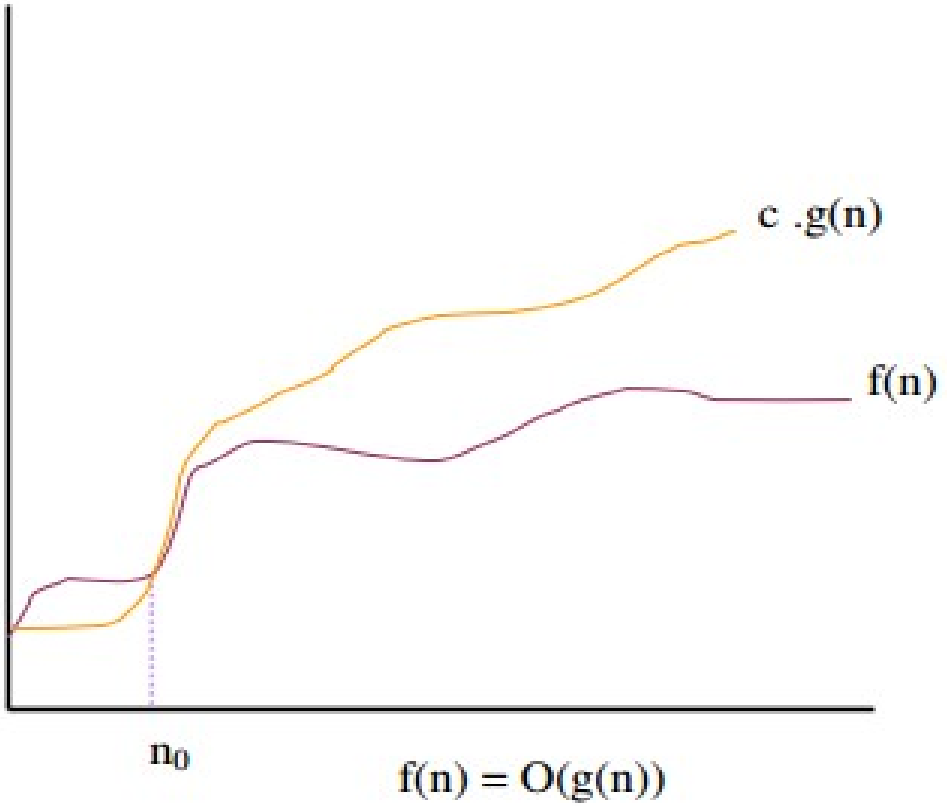
Big O Notation

- Big-O, commonly written as **O**, is an Asymptotic Notation for the worst case, or ceiling of growth for a given function.
- It provides us with an ***asymptotic upper bound*** for the growth rate of run-time of an algorithm.

Big O Notation

- Say **f(n)** is your algorithm run-time, and **g(n)** is an arbitrary time complexity you are trying to relate to your algorithm.
- **f(n)** is **O(g(n))**, if for some real constants **c (c > 0)** and $\mathbf{n_0}$, **f(n) ≤ c*g(n)** for every input size **n (n > n_0).**
- Big-O is the primary notation use for general algorithm time complexity.

Big O Notation



Big O Notation

Example 1

$f(n) = 3*n^2$

$g(n) = n$

Is f(n), O(g(n))?

- Let's look at the definition of Big-O.
- $3 * n^2 \leq c * n$
- Is there some pair of constants c, n_0 that satisfies this for all $n > 0$?
- No, there isn't.
- Thus,f(n) is NOT O(g(n)).

Big O Notation

Example 2

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$, then prove that $f(n) = O(g(n))$.

Proof: let us choose c and n_0 values as 14 and 1 respectively then we can have

$f(n) <= c*g(n), n>=n0$ as

$3n^2 + 4n + 7 <= 14*n^2$ for all $n >= 1$

the above inequality is trivially true

hence $f(n) = O(g(n))$

Big O Notation

Example 3

f(n) = 2n + 5

g(n) = n

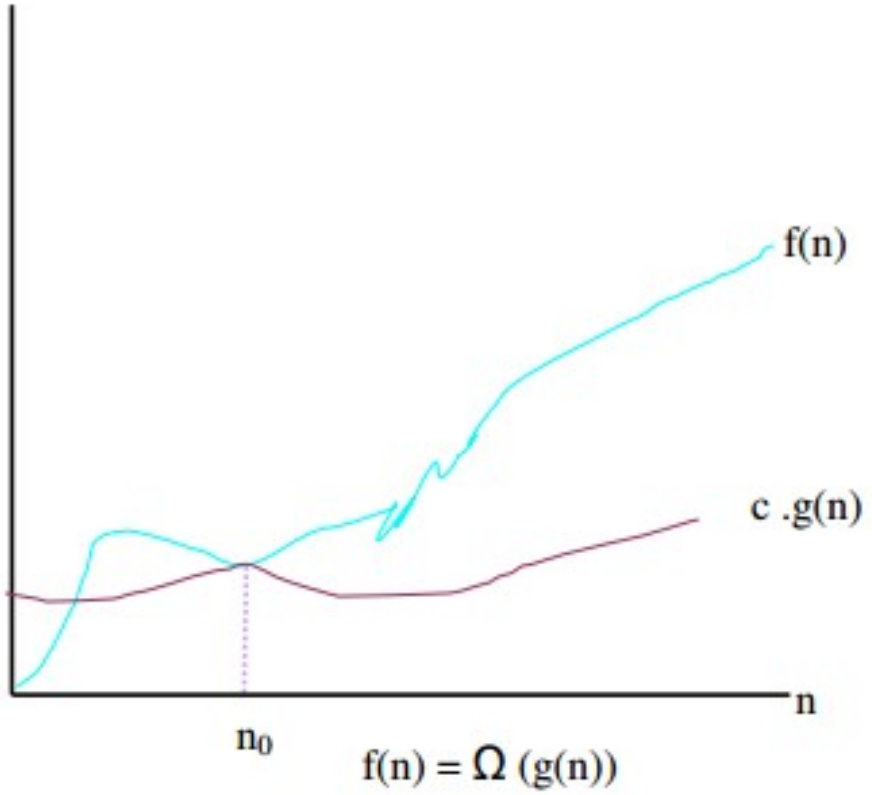
Is f(n), O(g(n))?

- 2n + 5 $\leq$ 7*n i.e c = 7 and this will be true for all n>0.
- f(n) is O(g(n))
- Thus, f(n) is O(n).

Big Ω Notations

- Big-Omega, commonly written as **Ω**, is an Asymptotic Notation for the best case, or a floor growth rate for a given function.
- It provides us with an ***asymptotic lower bound*** for the growth rate of run-time of an algorithm.
- **f(n)** is **Ω(g(n))**, if for some real constants **c (c > 0)** and **n_0 (n_0 > 0)**, **f(n)** ≥ **c *g(n)** for every input size **n (n > n_0).**

Big Ω Notations



$f(n) = \Omega(g(n))$

For all values of $n >= n_0$, plot shows clearly that f(n) lies above or on the curve of c*g(n).

Big Ω Notations

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$, then prove that $f(n) = \Omega(g(n))$.

Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have

$f(n) >= c*g(n), n>=n0$ as

$3n^2 + 4n + 7 >= 1*n^2$ for all $n >= 1$

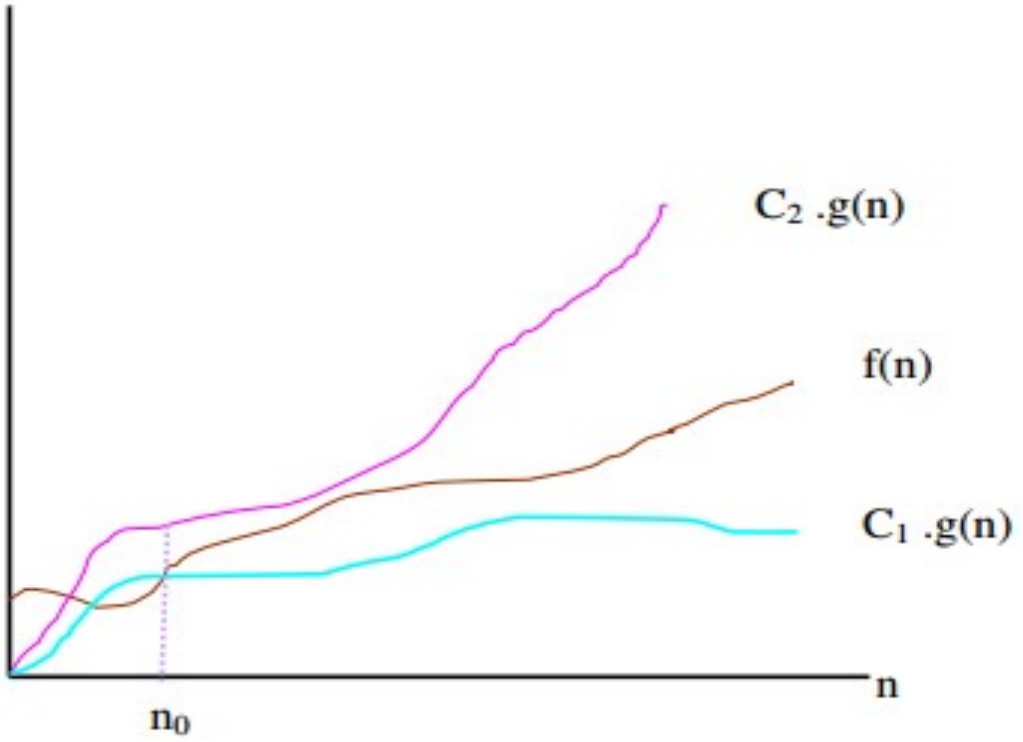
the above inequality is trivially true

hence $f(n) = \Omega(g(n))$

Big Θ Notations

- Theta, commonly written as **Θ**, is an Asymptotic Notation to denote the ***asymptotically tight bound*** on the growth rate of run-time of an algorithm.
- **f(n)** is **Θ(g(n))**, if for some real constants **c_1**, **c_2** and **n_0 ($c_1 > 0$, $c_2 > 0$, $n_0 > 0$)**, **$c_1 * g(n) < f(n) < c_2 * g(n)$** for every input size **n ($n > n_0$)**.
- f(n) is Θ(g(n)) implies f(n) is O(g(n)) as well as f(n) is Ω(g(n)).

Big Θ Notations



$f(n) = \Theta(g(n))$

For all values of $n >= n_0$, plot shows clearly that f(n) lies between c_1* g(n)and c_2*g(n).

Big Θ Notations

Example

f(n) = 2n + 5

g(n) = n

- f(n) = O(n) because 2n + 5 ≤ 7n for all n>0
- f(n) = Ω(n) because 2n + 5 ≥ n for all n>0
- 1*n < f(n) < 7*n for all n > 0
- Thus, f(n) is Θ (n)

Big Θ Notations

$f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$, then prove that $f(n) =$ $(g(n))$.

Proof: let us choose c1, c2 and n_0 values as 14, 1 and 1 respectively then we can have,

$f(n) <= c1*g(n), n>=n0$ as $3n^2 + 4n + 7 <= 14*n^2$, and

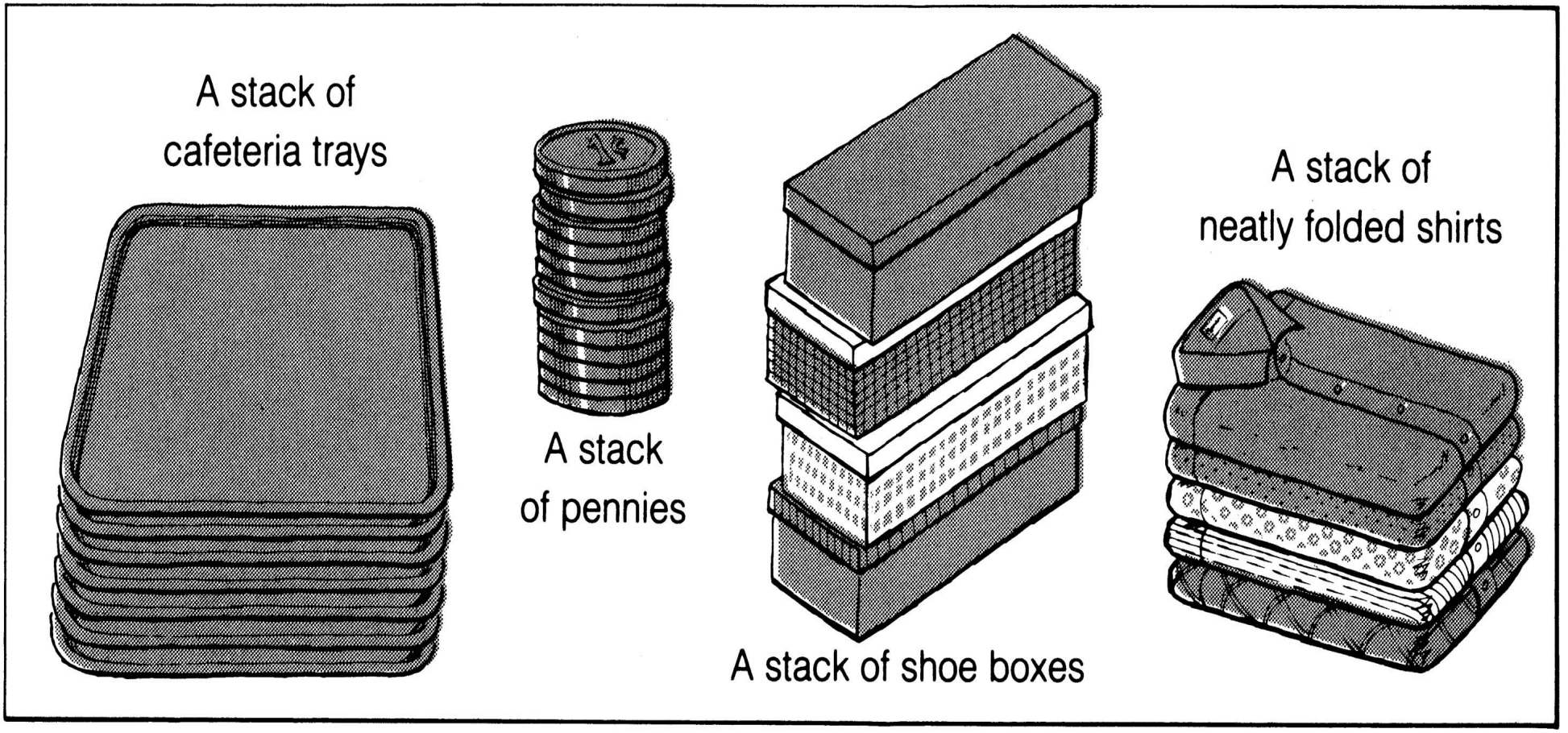
$f(n) >= c2*g(n), n>=n0$ as $3n^2 + 4n + 7 >= 1*n^2$

for all $n >= 1$(in both cases).

So $c2*g(n) <= f(n) <= c1*g(n)$ is trivial.

hence $f(n) = \Theta (g(n))$.

Stack


A stack of
cafeteria trays
A stack
of pennies
A stack of shoe boxes
A stack of
neatly folded shirts

Stack

- A stack is a linear data structure where items can be inserted and removed only at one end called top of stack such that items which are inserted at the last are removed first(LIFO).

Stack

What can we do with a stack?

- ✓ **push** - place an item on the stack.
- ✓ **pop** - Look at the item on top of the stack and remove it.
- ✓ **isFull** – Check if the stack is full?
- ✓ **isEmpty** – Check if the stack is empty?

Stack

```
isEmpty(S)
{
    if S.top == -1
        return TRUE
    else
        return FALSE
}
```

Stack

```
isFull(S)
{
    if S.top == MAX-1
        return TRUE
    else
        return FALSE
}
```

Stack

```
push(S, x)
{
    if isFull(S)
        Display “Overflow”
    else
        S.top = S.top + 1
        S[S.top] = x
}
```

Stack

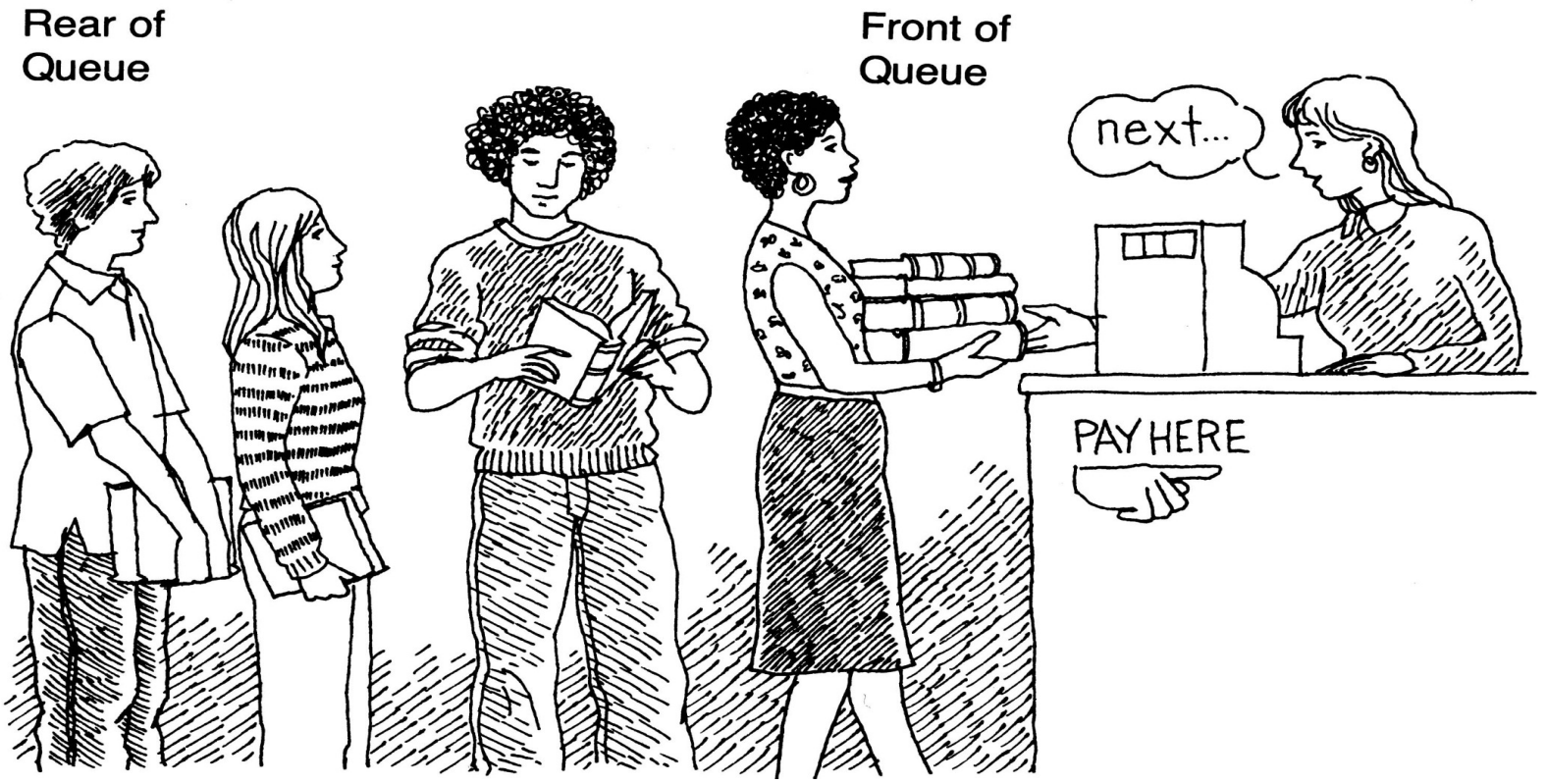
```
pop(S)
{
    if isEmpty(S)
        Display “Underflow”
    else
        S.top = S.top - 1
        return S[S.top + 1]
}
```

Stack

- Each of the aforementioned stack operations take O(1) time.

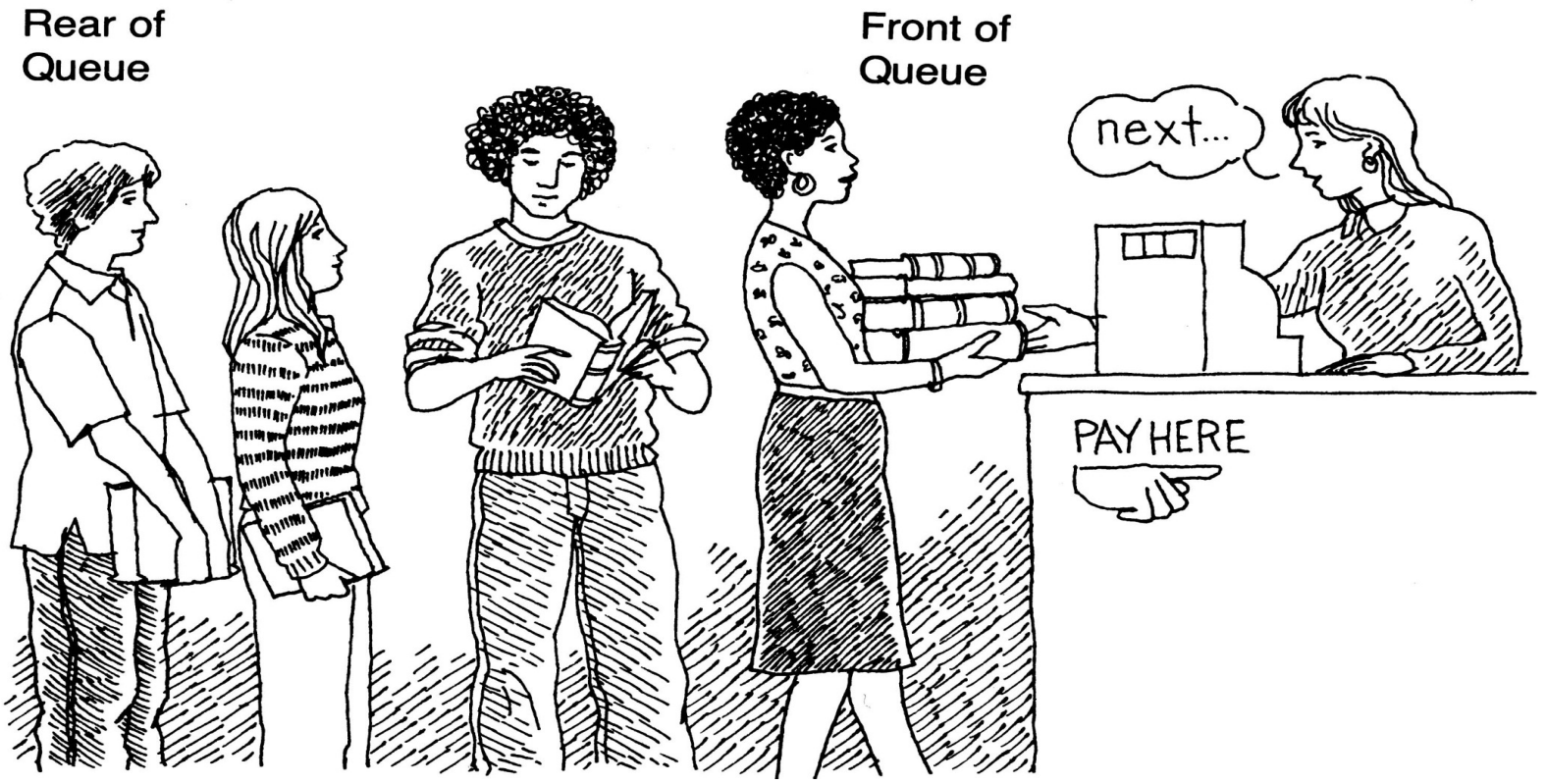
Queue

- A queue is a linear data structure which has two ends, FRONT and REAR.
- Elements are inserted at REAR and removed from front.



Queue

- The elements which is inserted first also gets removed first.
- This is also known as FIFO.



Queue

- Placing an item in a queue is called “insertion or enqueue”, which is done at the end of the queue called “rear”.
- Removing an item from a queue is called “deletion or dequeue”, which is done at the other end of the queue called “front”.

Queue

Enqueue(Q, x)

1) Start
2) Initialize Q.front=0 and Q.rear=-1
3) If Q.rear = MAX-1 then Display “Queue is Full”.
4) Else

    Q.rear = Q.rear + 1

    Q[Q.rear] = x
5) Stop

Queue

Dequeue(Q)

1) Start
2) If Q.rear < Q.front then Display “Queue is Empty”.
3) Else

    Q.front = Q.front + 1

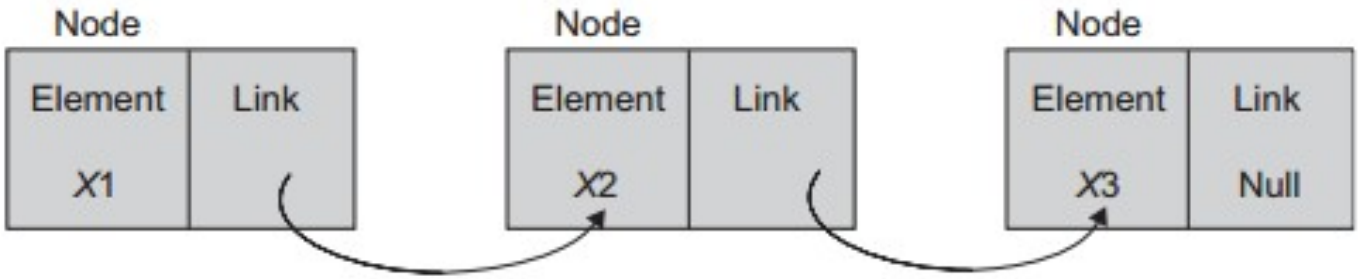
    return Q[Q.front - 1]
4) Stop

Queue

- Each of the aforementioned queue operations take O(1) time.

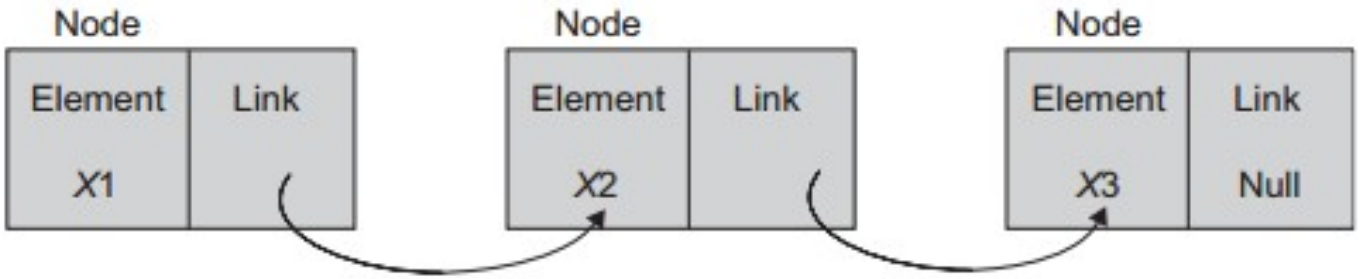
Linked Lists

- A linked list is a collection of data in which each element (node) contains a minimum of two values, data and link(s) to its successor (and/or predecessor).



Linked Lists

- In a linked list, before adding any element to the list, a memory space for that node must be allocated.
- A link is made from each item to the next item.



Linked Lists

Each node of the linked list has at least the following two elements:

1. The data member(s) being stored in the list.
2. A pointer or link to the next element in the list.

The last node in the list contains a null pointer (or a suitable value like -1) to indicate that it is the end.

Linked Lists - Features

- ✔ Dynamic allocation that is, space allocation as per need can be done during execution.
- ✔ As objects are not placed in consecutive locations at a fixed distance apart, random access to elements is not possible.
- ✔ Insertion and deletion of objects do not require any data shifting.

Linked Lists - Features

- It is space efficient for large objects with frequent insertions and deletions.
- Each element in general is a collection of data and a link. At least one link field is a must.
- Every element keeps the address of its successor(or predecessor)element in a link field.
- The only burden is that we need additional space for the link field for each element. However, additional space is not a severe penalty when large objects are to be stored.

Variations of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

Singly Linked List

- A SLL has two fields; data field and a link field.
- The link field is used to keep track of the successor.

Singly Linked List

```
struct SLL
{
   int info;
   struct SLL *next;
};
```

Singly Linked List – Operations

- Insertion
- Deletion
- Searching

Singly Linked List – Insertion

Can be done in many ways

- at the beginning
- at the end
- at specified position

Singly Linked List – Insertion at the Beginning

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Insertion at the Beginning

1. Start
2. Create a node using the malloc function as

   NewNode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say element
4. Assign data to the info field of the new node

   NewNode → info=element

   NewNode → next = NULL;

Singly Linked List – Insertion at the Beginning

5. If (first==null) then

   Set, first=last=NewNode and exit.

6. Else,Set next of new node to first

   NewNode→next=first;

7. Set the first pointer to the new node

   first=NewNode;

8. Stop

Singly Linked List – Insertion at the End

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Insertion at the End

1. Start
2. Create a node using the malloc function as

   NewNode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say element
4. Assign data to the info field of the new node

   NewNode → info=element

   NewNode → next = NULL;

Singly Linked List – Insertion at the End

5. If (first==null) then

   Set, first=last=NewNode and exit.

6. Else,Set next of last to NewNode

   last→next=NewNode;

7. Set the last pointer to the new node

   last=NewNode;

8. Stop

Singly Linked List – Insertion at the Specified Position

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Insertion at the Specified Position

1. Start
2. Create a node using the malloc function as

   NewNode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say element
4. Assign data to the info field of the new node

   NewNode → info=element

   NewNode → next = NULL;

Singly Linked List – Insertion at the Specified Position

5. Enter position of a node at which you want to insert a new node say pos.
6. Set, temp=first;
7. If (first==null) then

   Print “void insertion” and exit.
8. for(i=1;i<pos-1;i++)

   temp=temp → next;
9. Set, NewNode → next=temp → next
10. Set temp → next = NewNode
11. Stop

Singly Linked List – Deletion

Can be done in many ways

- from the beginning
- from the end
- from the specified position

Singly Linked List – Deletion from the Beginning

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Deletion from the Beginning

1. Start
2. If (first==null) then
   Print “void deletion” and exit;
3. Else if(first == last)
   Print deleted item as first→ info;
   first=last=null;
4. Else, Store the address of first node in temp
   temp=first;
5. Set first to next of first
   first=first → next
6. Free the memory of temp
   free(temp);
7. Stop

Singly Linked List – Deletion from the End

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Deletion from the End

1.Start

2.If (first==null) then

Print “void deletion” and exit;

3.Else if(first == last)

Print deleted item as first→ info;

first=last=null;

4. Else,

temp=first;

while(temp→next !=last)

temp=temp → next

temp → next=null;

last=temp;

5. free(temp)

6.Stop

Singly Linked List – Deletion from the Specific Position

- Let first and last are the pointer to the first node and last node in the current list respectively.

Singly Linked List – Deletion from the Specific Position

1.Start

2.If (first==null) then

Print “void deletion” and exit;

3.Else if(first == last)

Print deleted item as first→ info;

first=last=null;

4. Else

temp=first

for(i=1;i<pos-1;i++)

temp=temp → next

5. loc = temp → next

6. Print deleted item is loc → info

7. Set temp→next=loc → next

8. free(loc)

9. Stop

Singly Linked List – Searching

- Let first and last are the pointer to the first node and last node in the current list respectively.
- key is the value we are searching for in our linked list.

Singly Linked List – Searching

1. Start
2. Initialize flag = 0
3. If (first == null) then Display “List is Empty” and exit.
4. Else

    temp = first

    while(temp → next != null)

    if(temp → info == key)

        Display “Search Successful” and Set flag=1;

    temp=temp → next;
5. if(flag==0) then Display “Search Unsuccessful”
6. Stop

Self Study

- Analyze the insertions, deletions and searching operations in a SLL.

Binary Search Trees

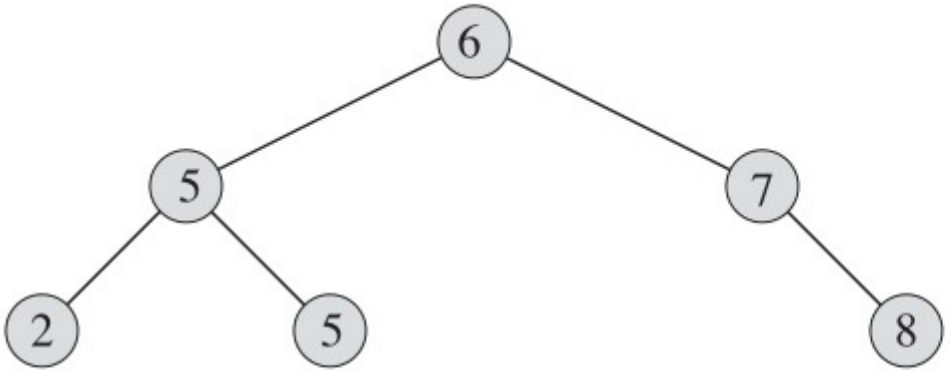
- A binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a binary tree data structure which has the following properties

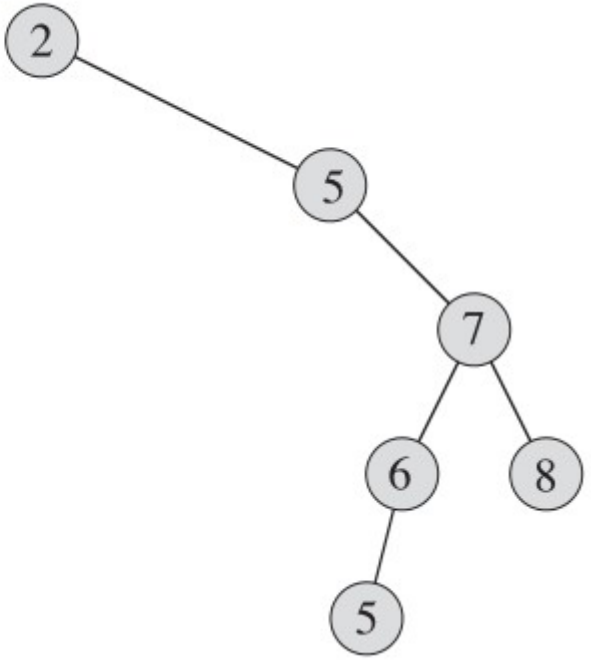
  - The left subtree of a node contains only nodes with keys less than or equal to the node's key.

  - The right subtree of a node contains only nodes with keys greater than or equal to the node's key.

  - The left and right subtree must each also be a binary search tree.

Binary Search Trees





Binary Search Trees

- The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property:

  Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key.

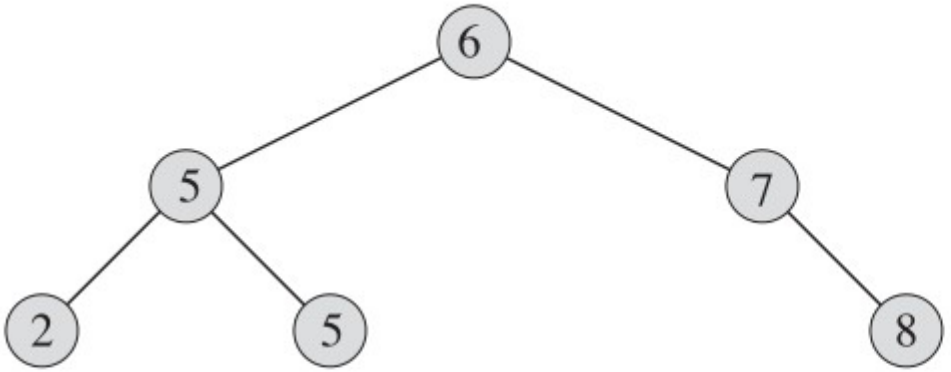
Binary Search Trees

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree walk.**
- This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree.

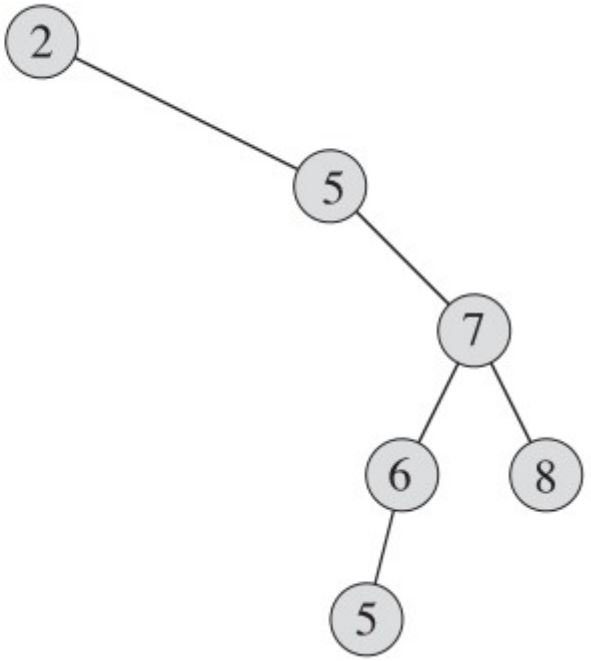
Binary Search Trees

```
INORDER-TREE-WALK(x)
{
    if x ≠ NIL
    {
        INORDER-TREE-WALK(x.left)
        Print x.key
        INORDER-TREE-WALK(x.right)
    }
}
```

Binary Search Trees



INORDER-TREE-WALK(x)
2, 5, 5, 6, 7, 8



INORDER-TREE-WALK(x)
2, 5, 5, 6, 7, 8

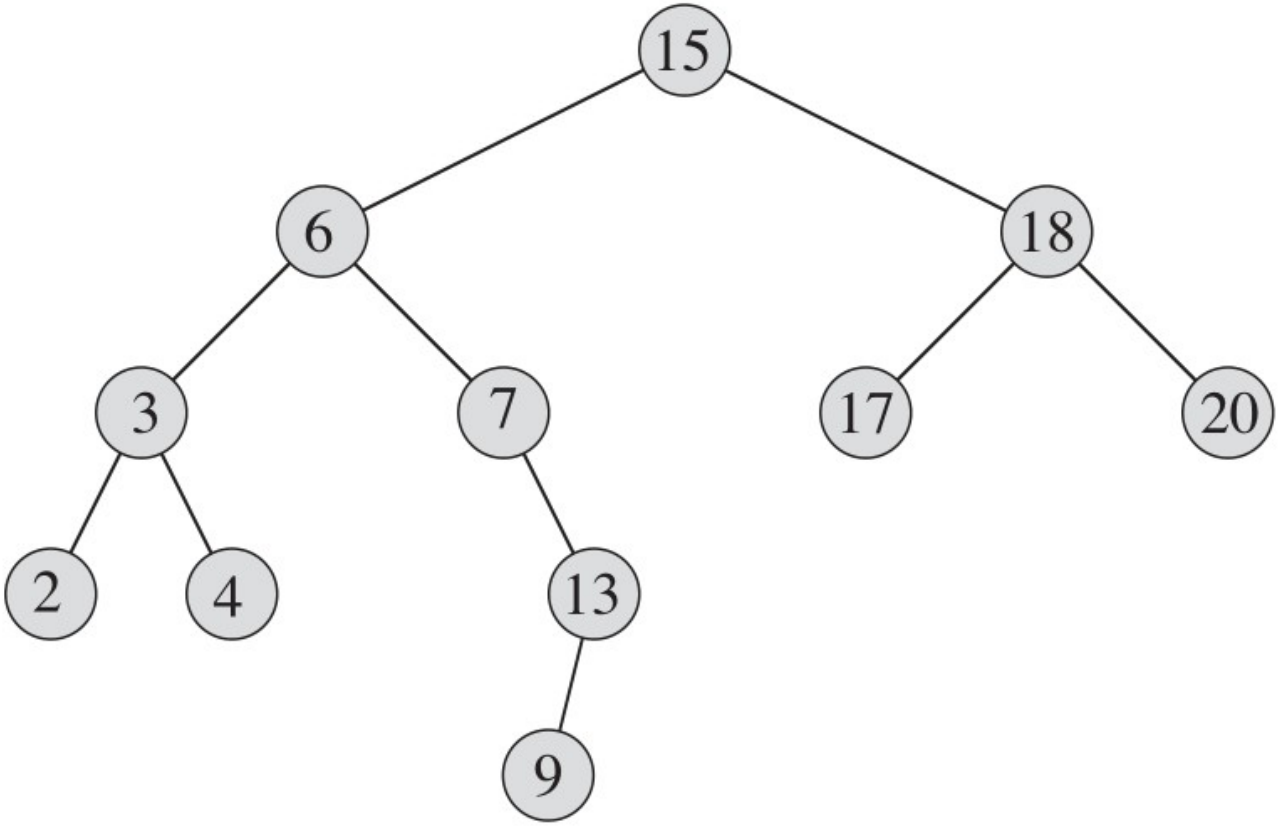
Binary Search Trees

- Similarly, a **preorder tree walk** prints the root before the values in either subtree, and a **postorder tree walk** prints the root after the values in its subtrees.

Binary Search Trees - Searching

- Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

Binary Search Trees - Searching



To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root.

Binary Search Trees - Searching

```
TREE-SEARCH(x, k)
{
    if x == NIL or k == x:key
        return x
    if k < x:key
        return TREE-SEARCH(x.left, k)
    else
        return TREE-SEARCH(x.right,k)
}
```

The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is O(h), where h is the height of the tree.

Binary Search Trees - Searching

ITERATIVE-TREE-SEARCH(x, k)

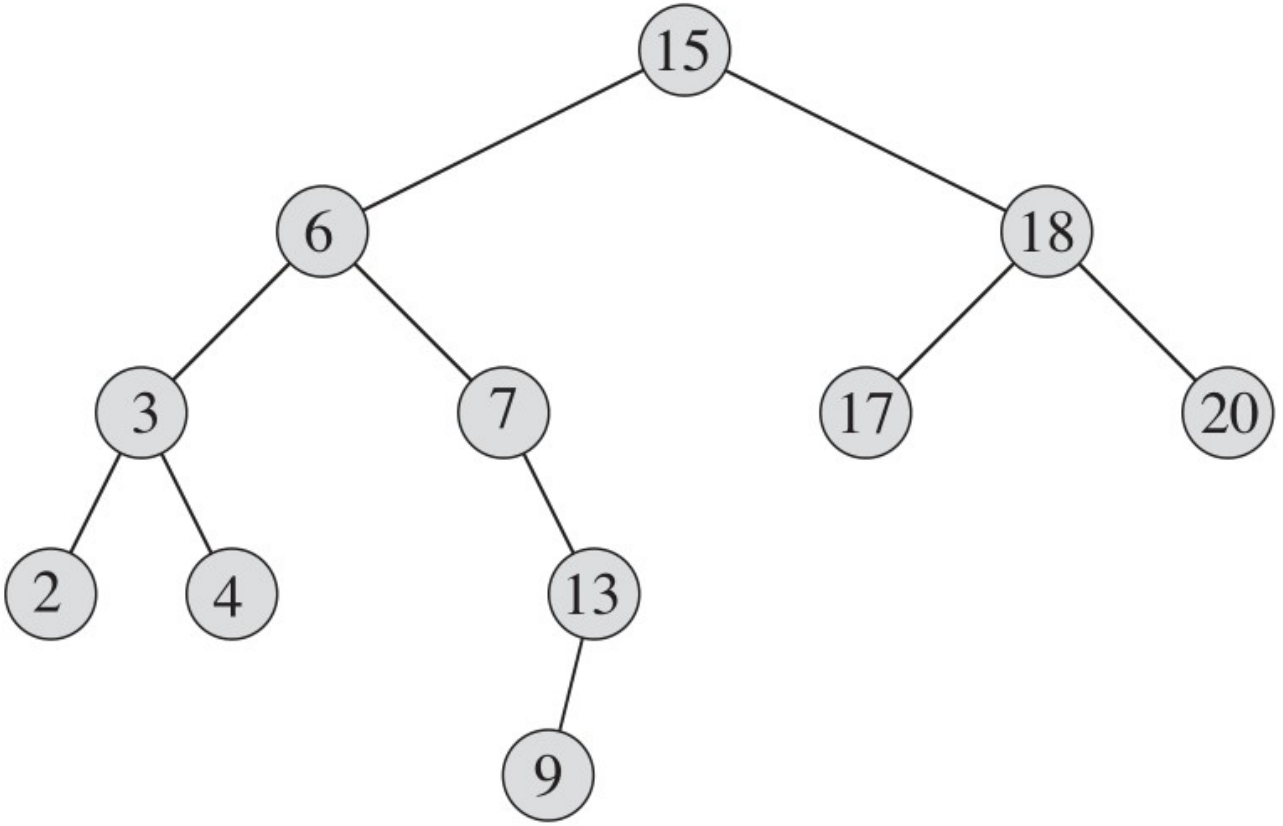
```
{
    while x ≠ NIL and k ≠ x.key
    {
        if k < x.key
            x = x.left
        else
            x = x.right
    }
    return x
}
```

The nodes encountered during this iterative version form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is O(h), where h is the height of the tree.

Binary Search Trees - Minimum

- We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a NIL.

Binary Search Trees - Minimum



The minimum key in the tree is 2, which is found by following left pointers from the root.

Binary Search Trees - Minimum

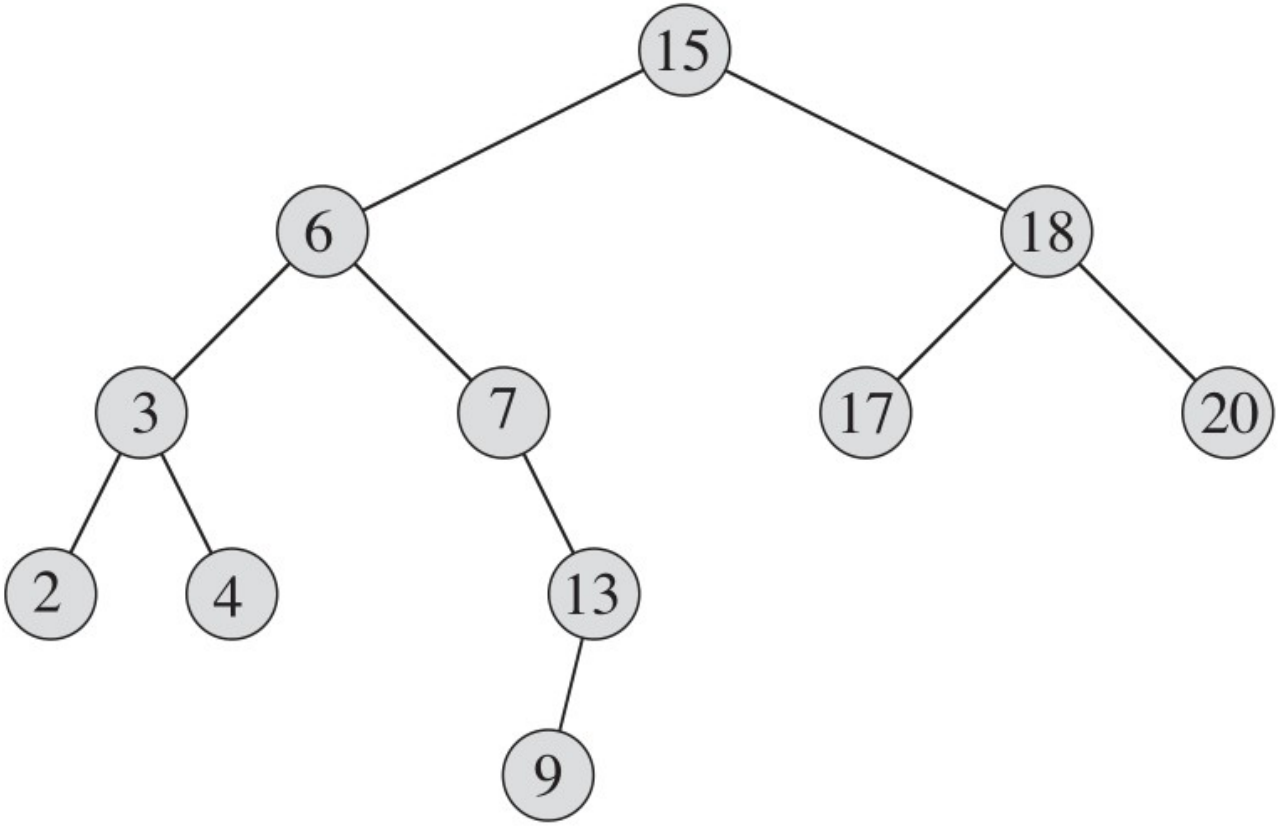
```
TREE-MINIMUM(x)
{
    while x.left ≠ NIL
        x = x.left
    return x
}
```

This procedure runs in O(h) time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Binary Search Trees - Maximum

- Similarly, we can always find an element in a binary search tree whose key is a maximum by following right child pointers from the root until we encounter a NIL.

Binary Search Trees - Maximum



The maximum key 20 is found by following right pointers from the root.

Binary Search Trees - Maximum

```
TREE-MAXIMUM(x)
{
    while x.right ≠ NIL
        x = x.right
    return x
}
```

This procedure runs in O(h) time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Binary Search Trees - Insertion

- To insert a new value Z into a binary search tree T , we use the procedure TREE- INSERT.
- The procedure takes a node Z for which Z.key = v and Z.left = NIL, and Z.right = NIL.
- It modifies T and some of the attributes of Z in such a way that it inserts Z into an appropriate position in the tree.

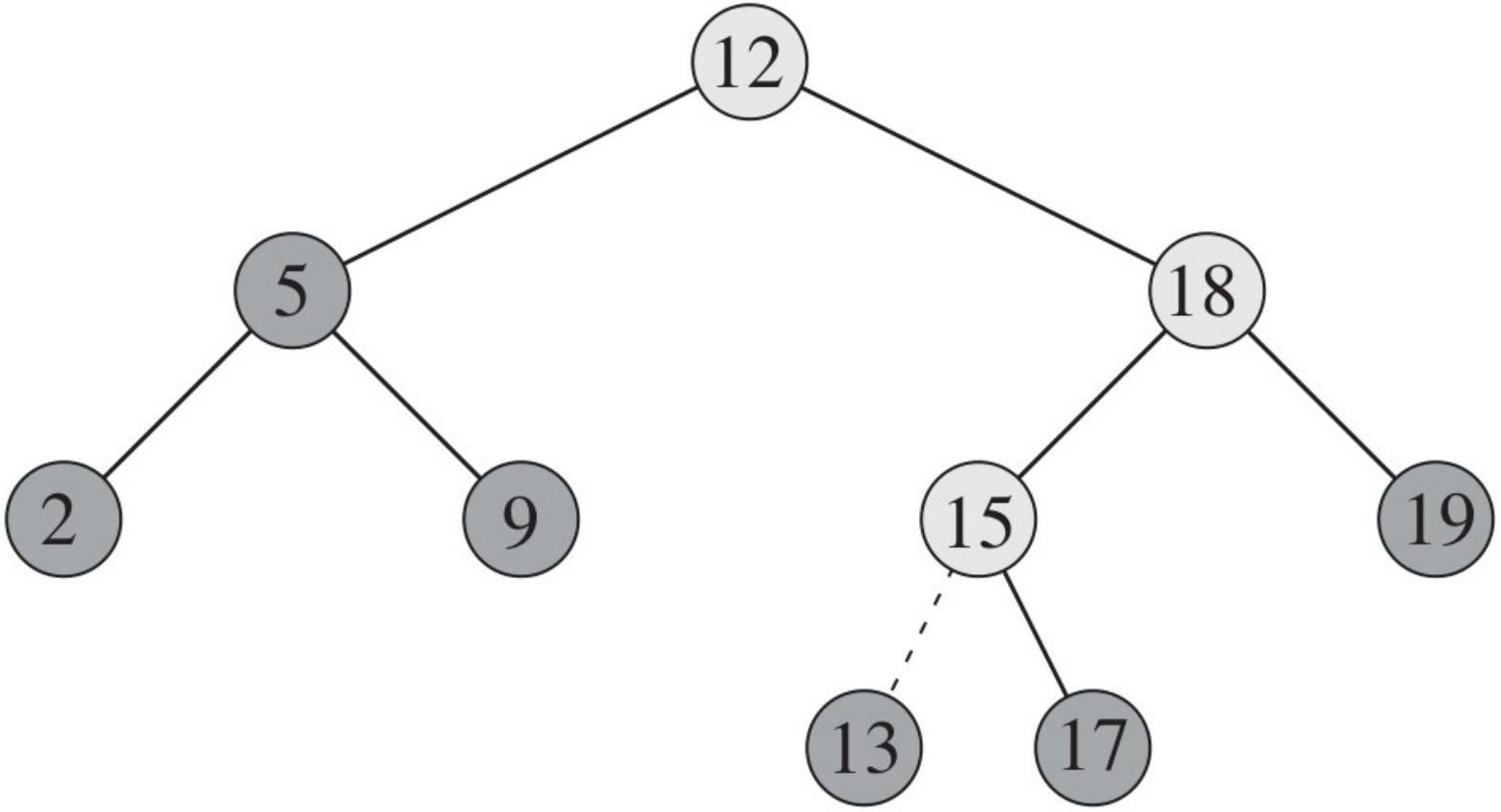
Binary Search Trees - Insertion

```
TREE-INSERT(T, Z)
{
y = NIL
x = T.root
while x ≠ NIL
{
    y = x
    if Z.key < x:key
        x = x.left
    else
        x = x.right
}´
```

```
Z.p = y

if y == NIL
    T.root = Z // tree T was empty

else if Z.key < y.key
    y.left = Z

else
    y.right =  Z
```

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in O(h)time on a tree of height h.

Binary Search Trees - Insertion



Binary Search Trees - Deletion

The overall strategy for deleting a node Z from a binary search tree T has three basic cases

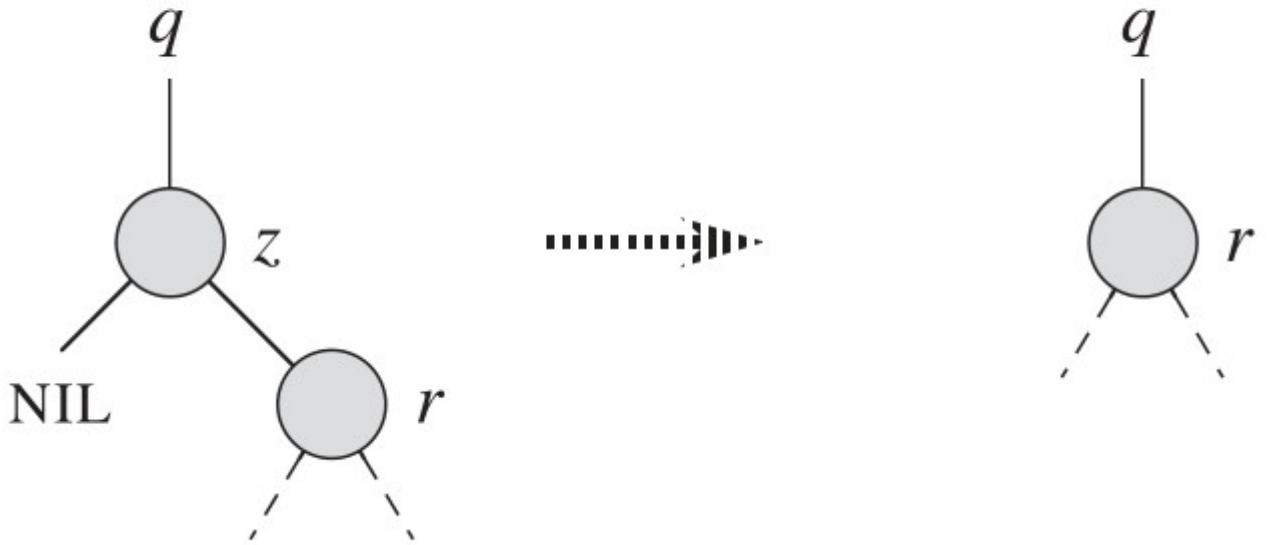
1. If Z has no children, then we simply remove it by modifying its parent to replace Z with NIL as its child.
2. If Z has just one child, then we elevate that child to take Z's position in the tree by modifying Z's parent to replace Z by Z's child.
3. If Z has two children, then we find Z's successor y—which must be in Z's right subtree—and have y take Z's position in the tree. The rest of Z's original right subtree becomes y's new right subtree, and Z's left subtree becomes y's new left subtree. This case is the tricky one because, as we shall see, it matters whether y is Z's right child.

Binary Search Trees - Deletion

- The procedure for deleting a given node Z from a binary search tree T takes as arguments pointers to T and Z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases.

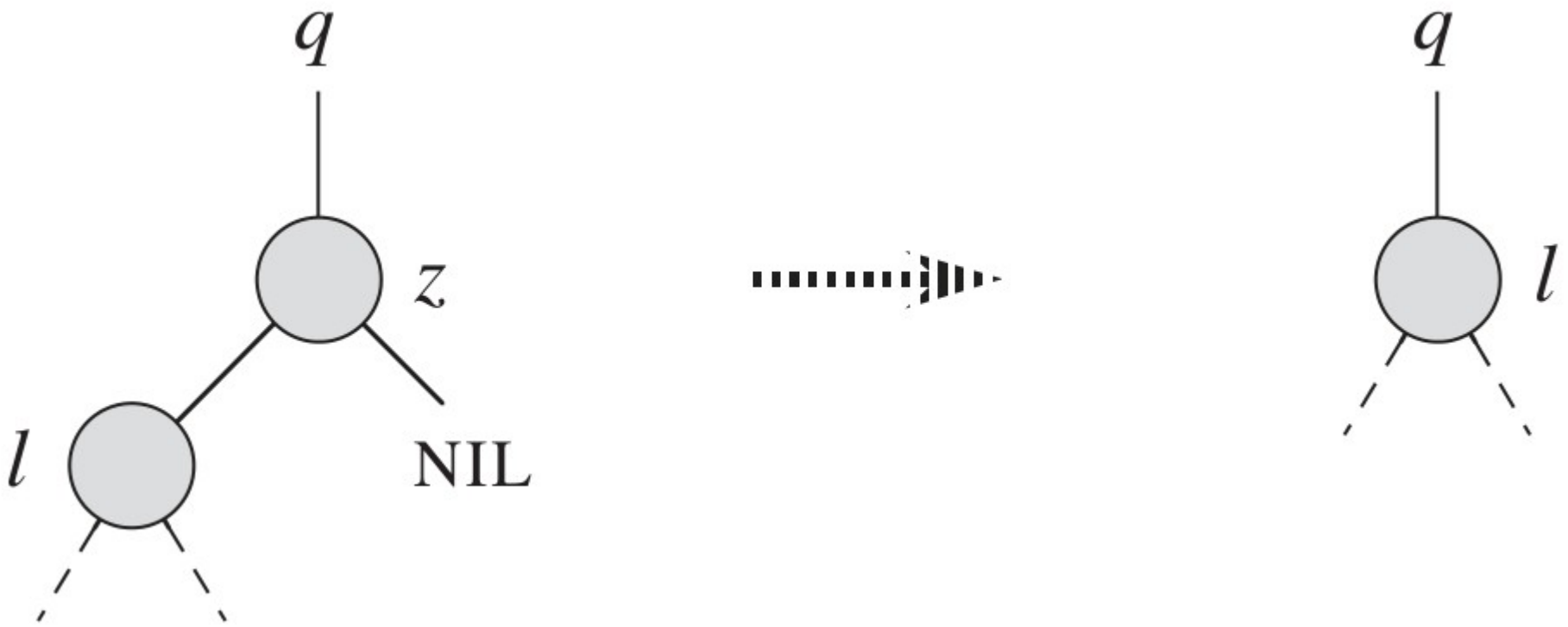
Binary Search Trees - Deletion

1. If Z has no left child , then we replace Z by its right child, which may or may not be NIL. When Z's right child is NIL, this case deals with the situation in which Z has no children. When Z's right child is non-NIL, this case handles the situation in which Z has just one child, which is its right child.



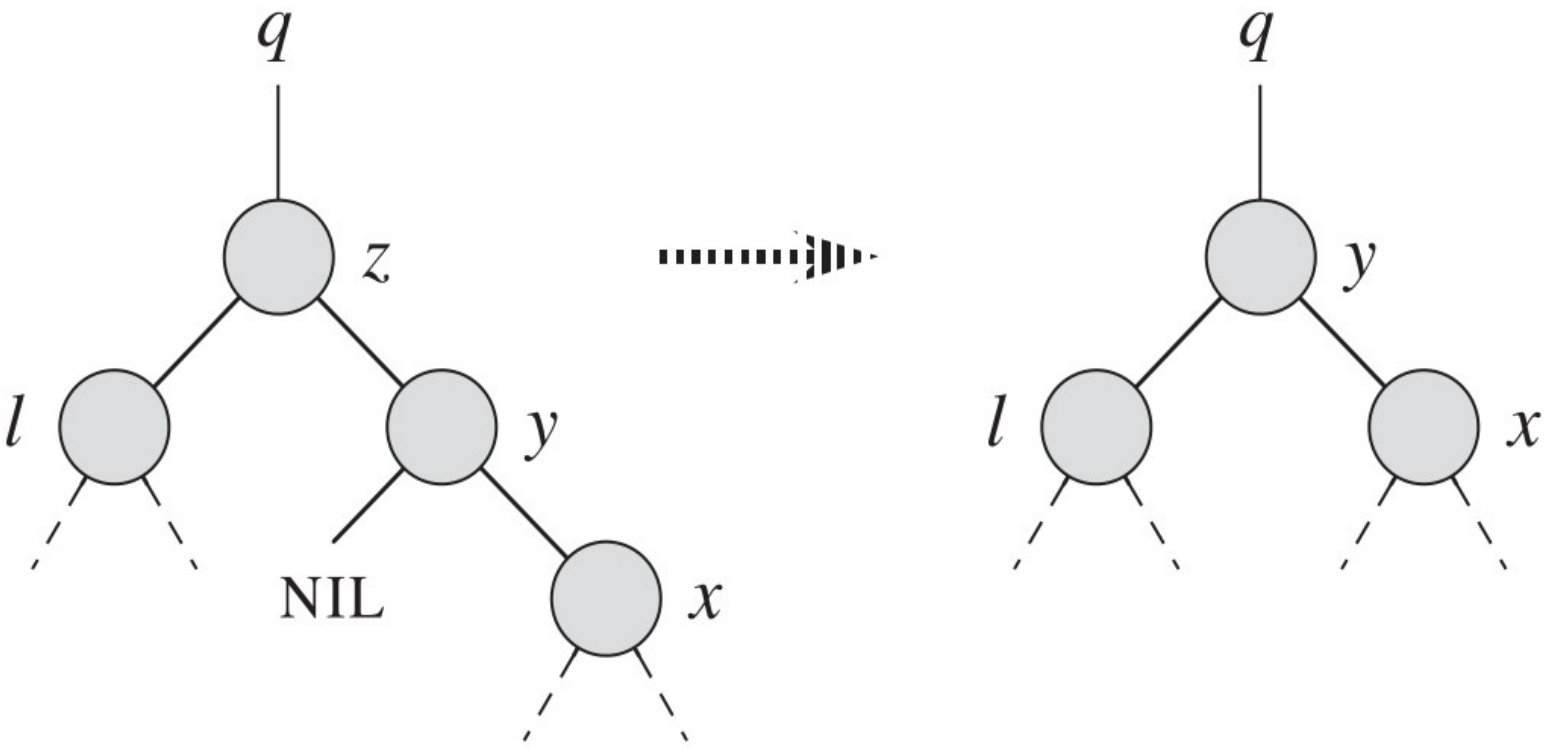
Binary Search Trees - Deletion

2. If Z has just one child, which is its left child, then we replace Z by its left child.



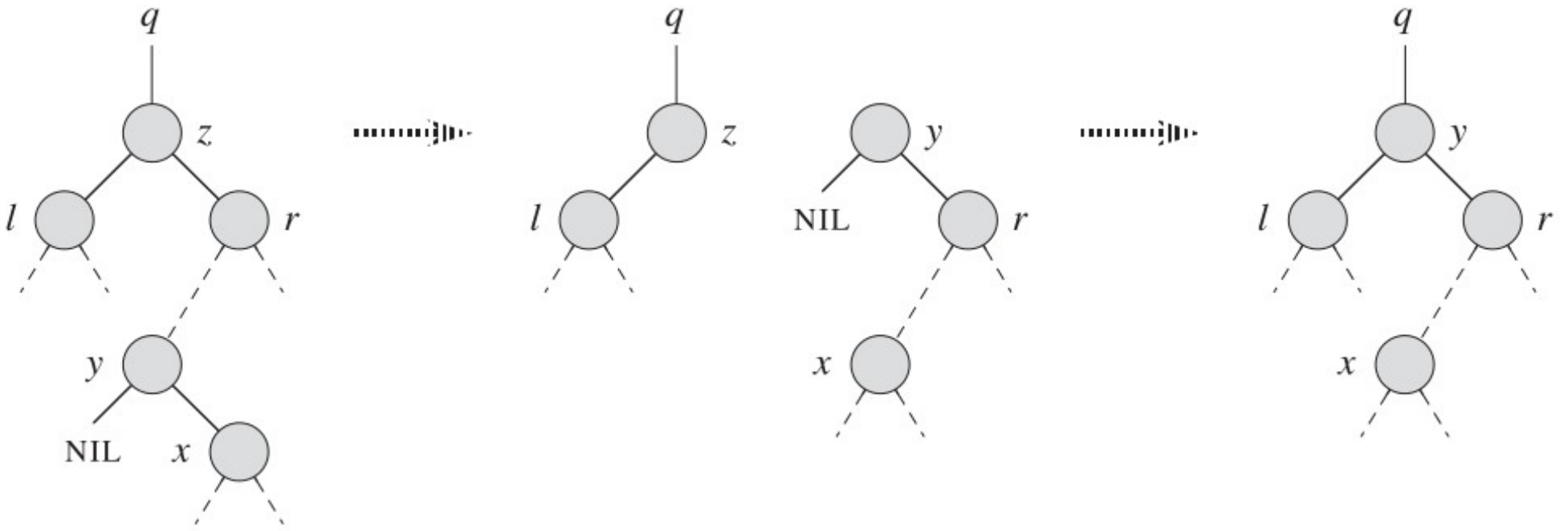
Binary Search Trees - Deletion

3. Otherwise, Z has both a left and a right child. We find Z's successor y, which lies in Z's right subtree and has no left child. **If y is Z's right child , then we replace Z by y, leaving y's right child alone.**



Binary Search Trees - Deletion

4. Otherwise, Z has both a left and a right child. We find Z's successor y, which lies in Z's right subtree and has no left child. **Otherwise, y lies within Z's right subtree but is not Z's right child. In this case, we first replace y by its own right child, and then we replace Z by y.**



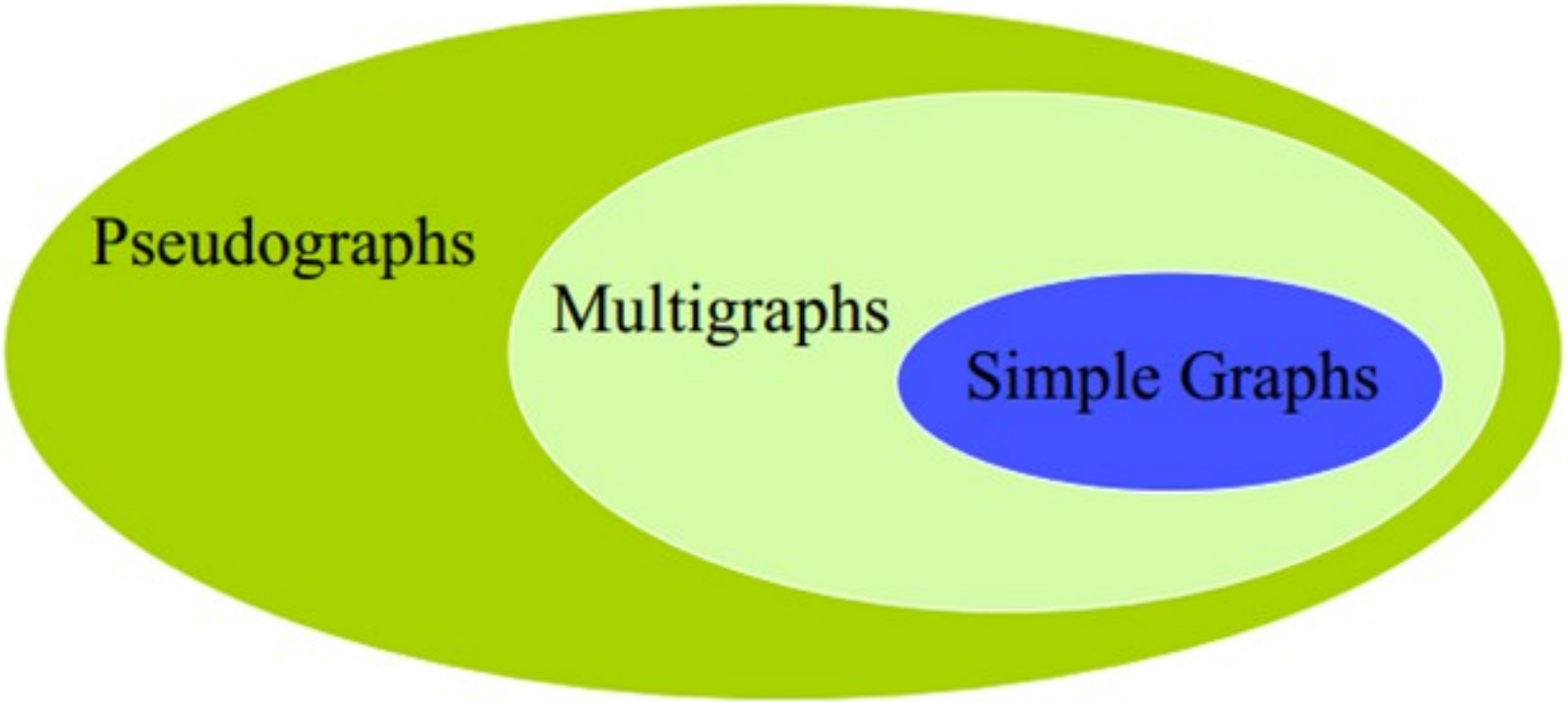
Self Study

- Write an algorithm to delete a node in BST and analyze it.

Graphs

- **Definition:** A simple graph G = (V, E) consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges.
- For each $e \in E$, e = {u, v} where u, $v \in V$.
- An undirected graph (not simple) may contain loops. An edge e is a loop if e = {u, u} for some $u \in V$.

Graphs - Types



Graphs Representation

- We can choose between two standard ways to represent a graph G =(V, E) as a collection of **adjacency lists** or as an **adjacency matrix.**
- Either way applies to both directed and undirected graphs.
- Because the adjacency-list representation provides a compact way to represent sparse graphs—those for which it is usually the method of choice.
- We may prefer an adjacency-matrix representation, however, when the graph is dense.

Graphs Representation

- **Definition:** Let G = (V, E) be a simple graph with |V| = n. Suppose that the vertices of G are listed in arbitrary order as v_1, v_2, …, v_n.
- The **adjacency matrix A** (or $\mathbf{A_G}$) of G, with respect to this listing of the vertices, is the n x n matrix with

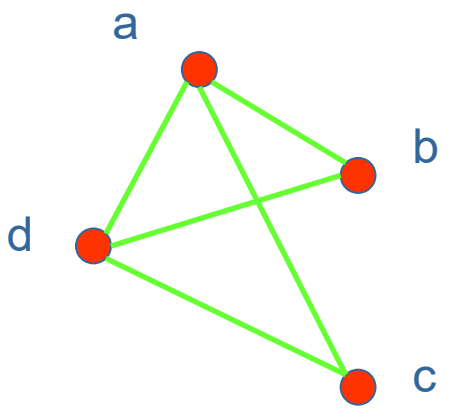
  a_{ij} = 1 if {vi, vj} is an edge of G,

  a_{ij} = 0 if there is no edge and

  a_{ij} = k if there are k($\geq$2) edges between the vertices.

Graphs Representation

•**Example:** What is the **adjacency matrix** A_G for the following graph G based on the order of vertices a, b, c, d ?



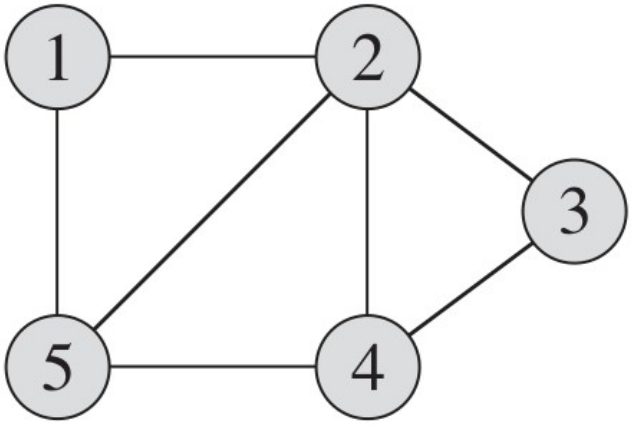
Solution:

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Note: Adjacency matrices of undirected graphs are always symmetric.

Graphs Representation

•**Example:** What is the **adjacency matrix** $\mathbf{A_G}$ for the following graph G based on the order of vertices 1, 2, 3, 4, 5 ?



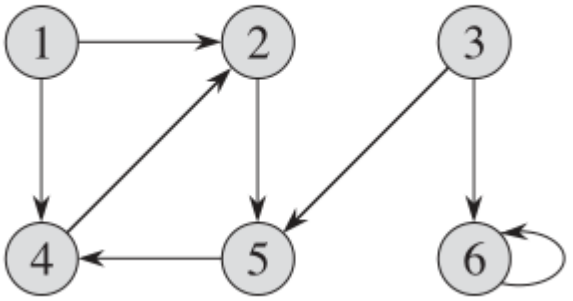
Solution:

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Note: Adjacency matrices of undirected graphs are always symmetric.

Graphs Representation

•**Example:** What is the **adjacency matrix** $\mathbf{A_G}$ for the following graph G based on the order of vertices 1, 2, 3, 4, 5, 6 ?



Solution:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Note: Adjacency matrices of undirected graphs are always symmetric.

Graphs Representation

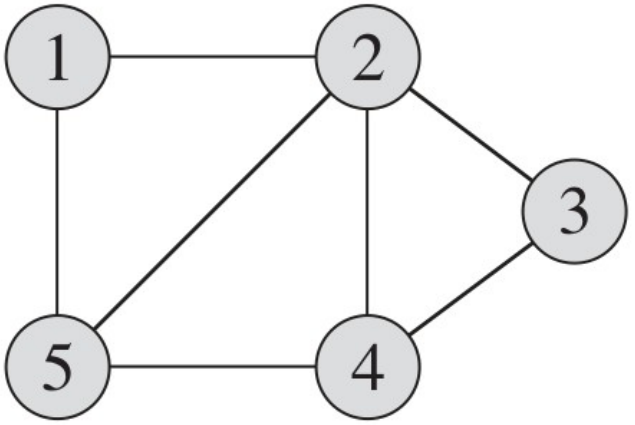
- The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.
- Adjacency matrices are simpler, and so we may prefer them when graphs are reasonably small.

Graphs Representation

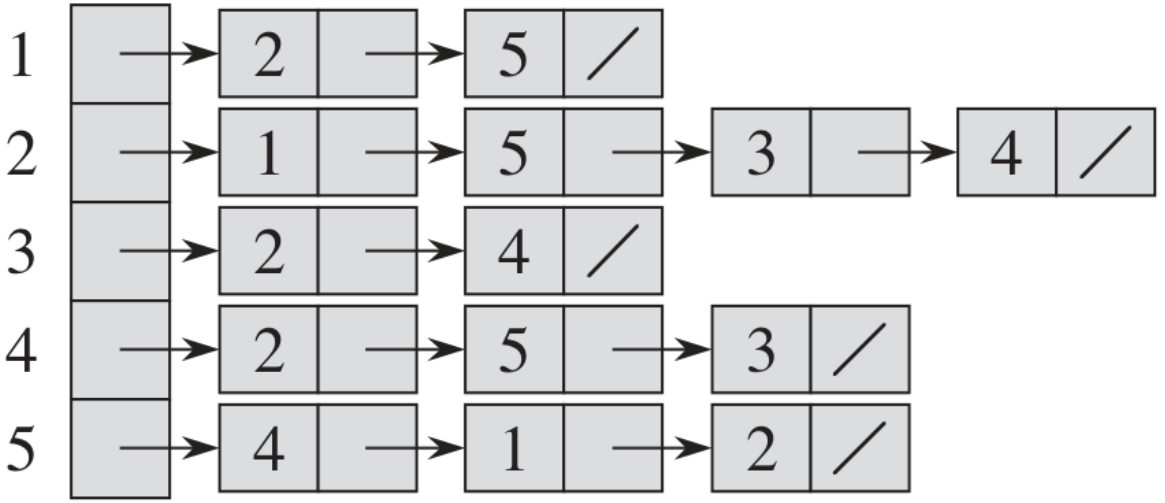
- The adjacency-list representation of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V.
- For each $u \in V$, the adjacency list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$.
- That is, Adj[u] consists of all the vertices adjacent to u in G.

Graphs Representation

•**Example:** What is the **adjacency list f**or the following graph G?

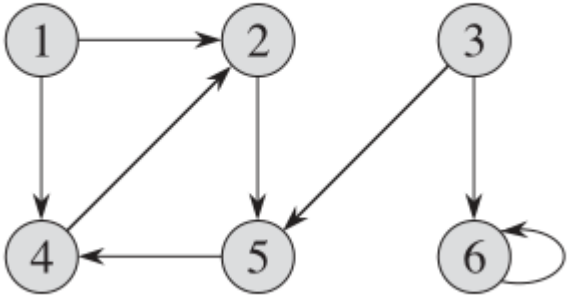


Solution:

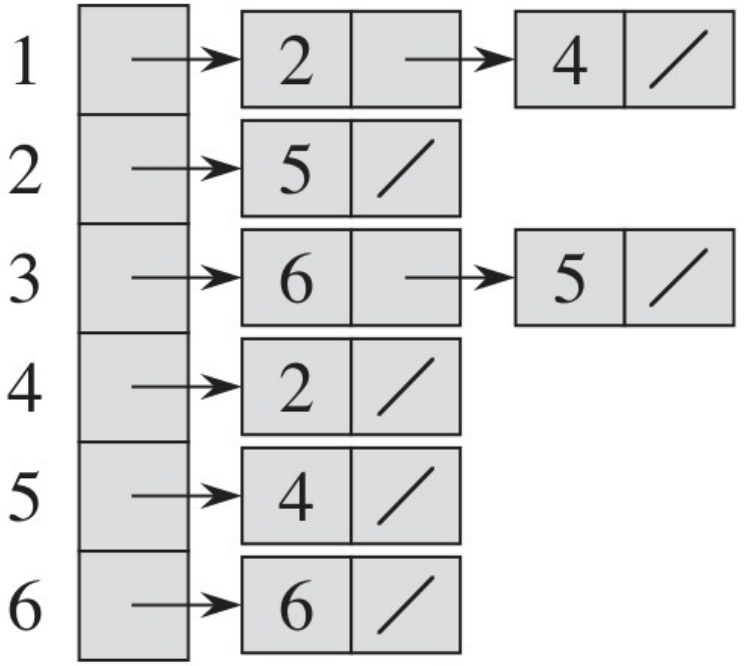


Graphs Representation

Example: What is the **adjacency list** for the following graph G?



Solution:



Breadth-First Search

- Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s.
- It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a “breadth-first tree” with root s that contains all reachable vertices.

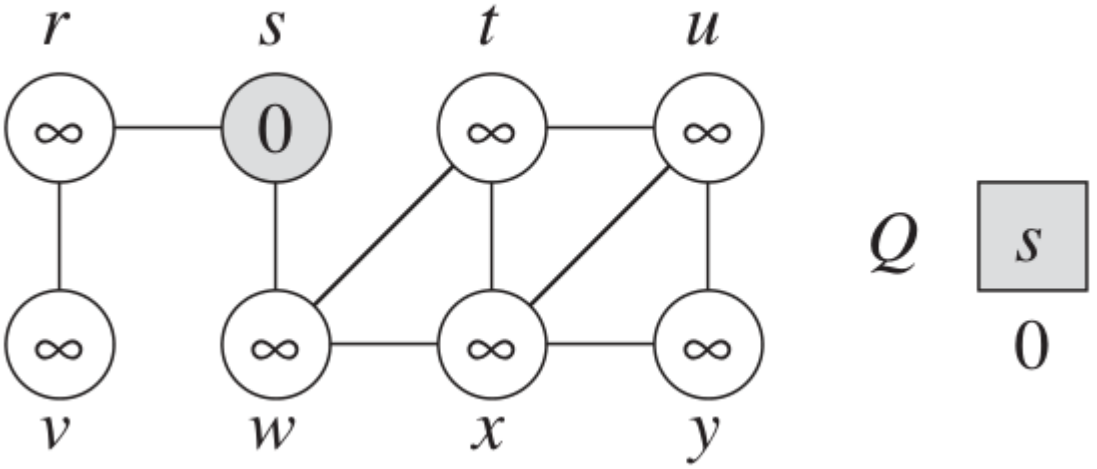
Breadth-First Search

- For any vertex v reachable from s, the simple path in the breadth-first tree from s to v corresponds to a “shortest path” from s to v in G, that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.
- Breadth-first search is named so because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance k +1.

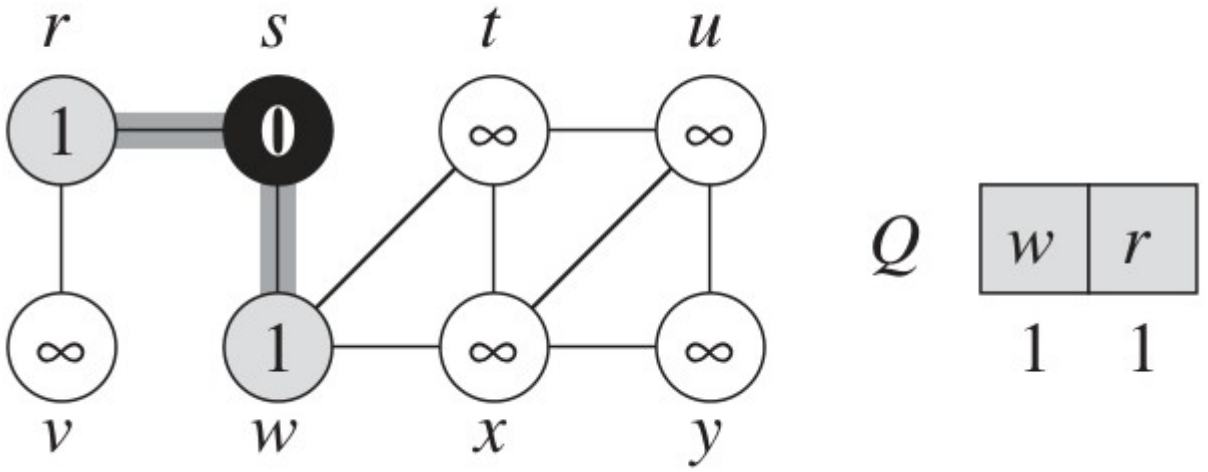
Breadth-First Search

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

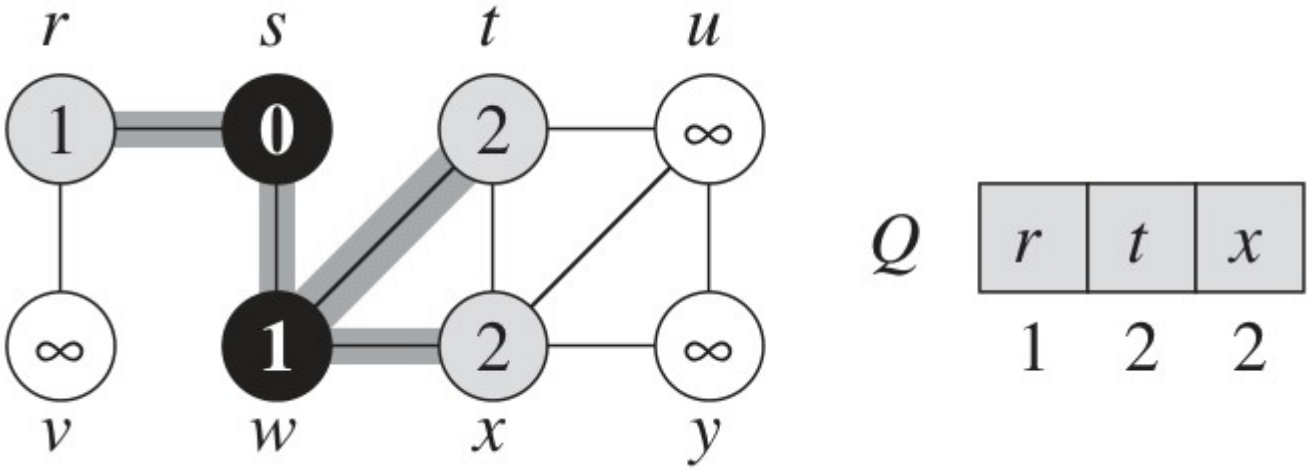
Breadth-First Search



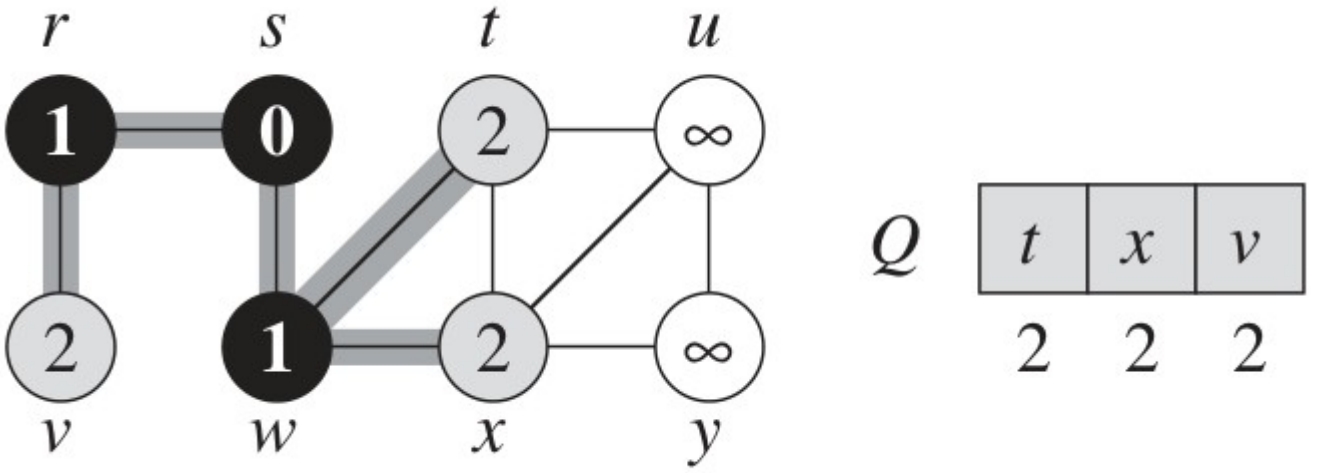
Breadth-First Search



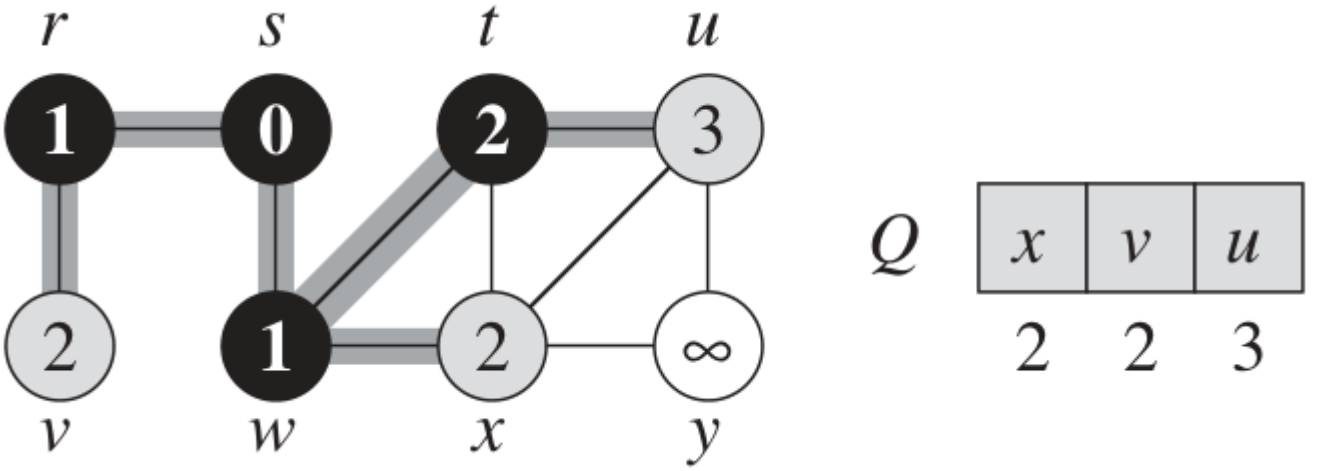
Breadth-First Search



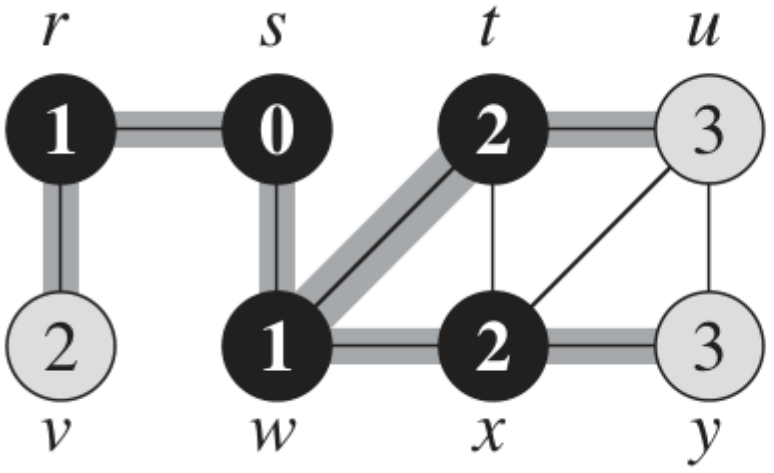
Breadth-First Search

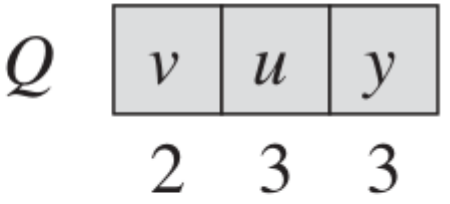


Breadth-First Search

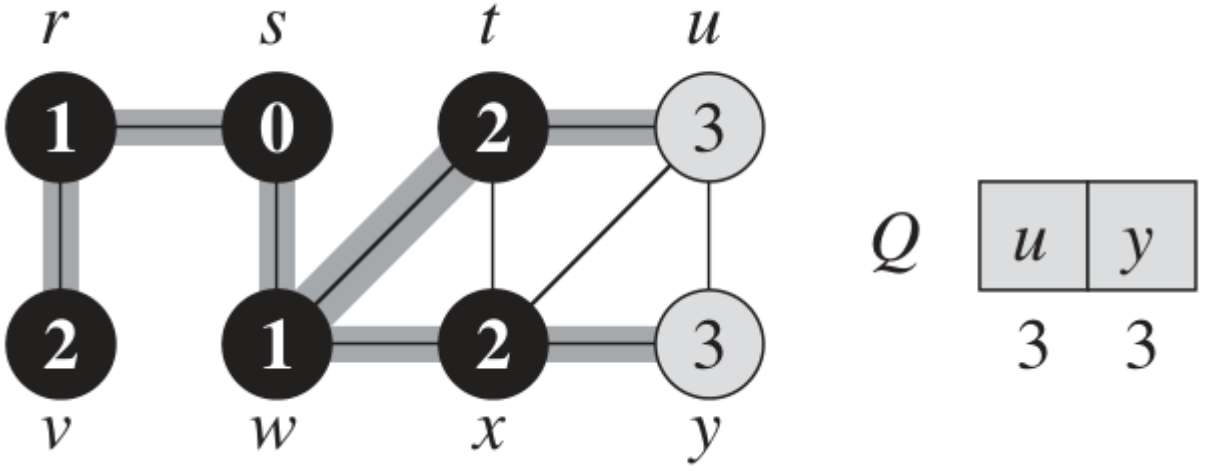


Breadth-First Search

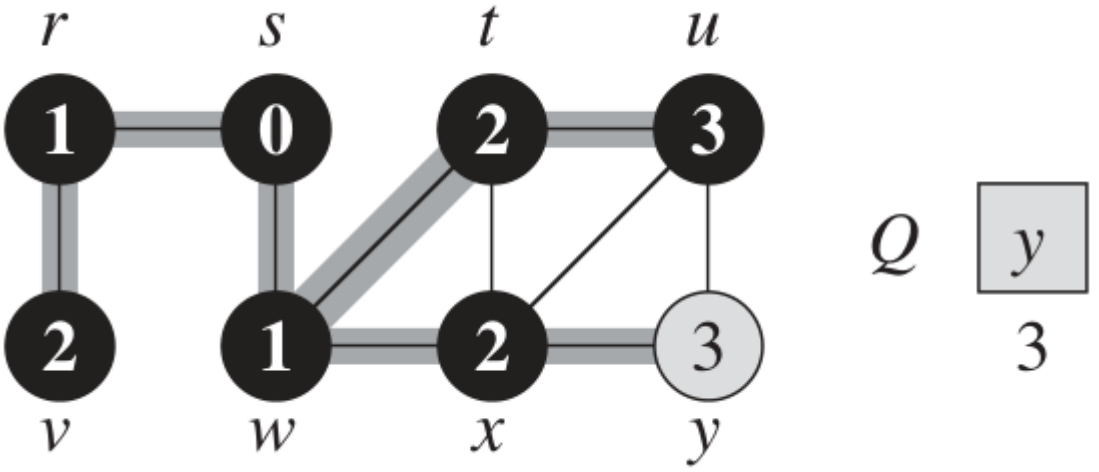




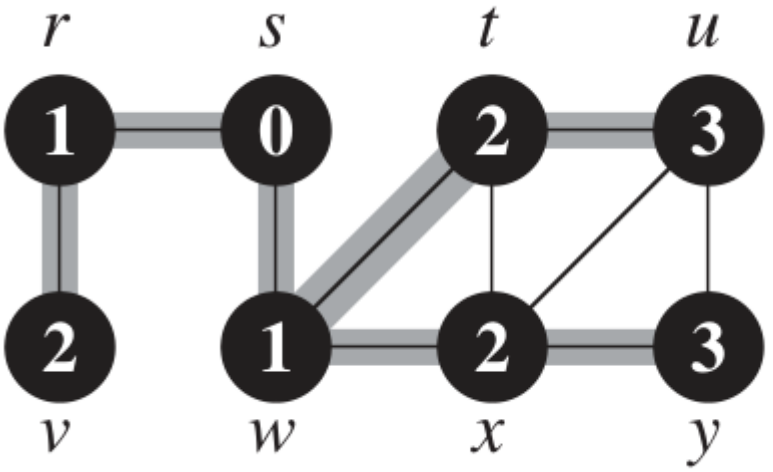
Breadth-First Search



Breadth-First Search



Breadth-First Search



Breadth-First Search - Analysis

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

- After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.
- The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V).

Breadth-First Search - Analysis

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

- Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
- Since the sum of the lengths of all the adjacency lists is Θ(E), the total time spent in scanning adjacency lists is O(E).

Breadth-First Search - Analysis

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

- The overhead for initialization is O(V), and thus the total running time of the BFS procedure is O(V+E).
- Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Depth-First Search

Self Study

Review Questions

1) Define big oh, big omega and big theta notations with suitable examples.
2) Find big oh and big omega of the following function

$$f(x) = 5n^3 + 6n^2 + 9n + 3$$

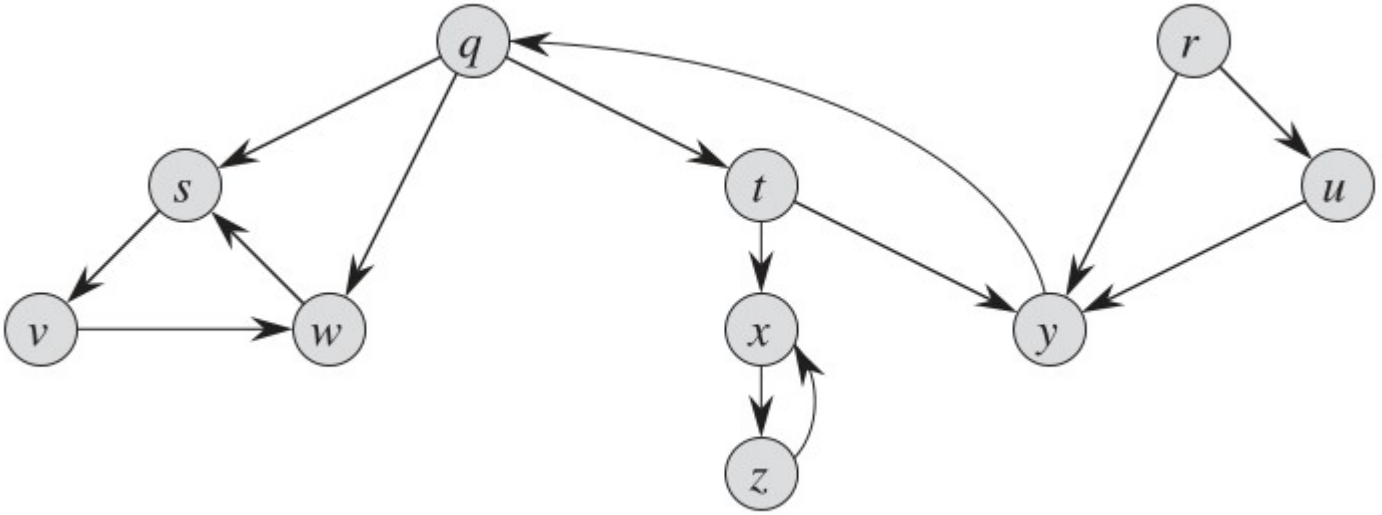
3) Explain why the statement, “The running time of algorithm A is at least $O(n^2)$,” is meaningless.
4) Illustrate the result of each operation in the sequence PUSH(S,4),PUSH(S, 1), PUSH(S,3), POP(S),PUSH(S,8), and POP(S) on an initially empty stack S stored in array S[0..5].
5) Explain and analyze the different operations in a stack.
6) Explain and analyze the different operations in a queue.
7) Implement a stack using a singly linked list L. The operations PUSH and POP should still take O(1) time.
8) Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take O(1) time.

Review Questions

9) Give recursive algorithms that perform preorder and postorder tree walks in Θ(n) time on a tree of n nodes.

10) Explain and analyze the different operations in a BST.

11) Show how depth-first search works on the graph of figure.



12) Explain and analyze breadth-first search in detail.