UNIT 1

Introduction to Data Structures

Lecture 1
What is a Data Structure?

- A *primitive data type* holds a single piece of data
  - e.g. in Java: int, long, char, boolean etc.
  - *Legal operations* on integers: + - * / ...
- A *data structure* structures data!
  - Usually more than one piece of data
  - Should provide legal operations on the data
  - The data might be joined together (e.g. in an array): a *collection*
- An *Abstract Data Type* (ADT) is a data type together with the operations, whose properties are specified independently of any particular implementation.

Principles of Good Design: Abstraction, Encapsulation, Modularity

ADTs use the following principles:

- Encapsulation: Providing data and operations on the data
- Abstraction: hiding the details.
  - e.g. A class exhibits what it does through its methods; however, the details of how the methods work is hidden from the user
- Modularity: Splitting a program into pieces.
  - An object-oriented program is a set of classes (data structures) which work together.
  - There is usually more than one way to split up a program

Principles of Good Design: High Cohesion, Low Coupling

- Modules (i.e. classes) should be as independent as possible
  - *Cohesion*: The extent to which methods in a class are related
  - *Coupling*: The extent to which a class uses other classes
  - Strive for **high cohesion** and **low coupling**
- The ADTs we will examine have high cohesion and low coupling

Basic Data Structures: Data Collections

- Linear structures
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back and remove from front
  - Priority queue: Add anywhere, remove the highest priority
- Hash tables: Unordered lists which use a ‘hash function’ to insert and search
- Tree: A branching structure with no loops
- Graph: A more general branching structure, with less stringent connection conditions than for a tree

Kinds of Operations

- Builders
  - Change the contents of the data structure
- Viewers
  - Retrieve the contents of the data structure
- Queries
  - Return information about the data structure
- Iterators
  - Return each element of the data structure, in some order

Lecture 2
Elementary Data Structures

Elementary Data Structure are fundamental approaches to organizing data. These are the building blocks that will be used to implement more complex Abstract Data Types.

1. Scalar (built-in) data types
2. Arrays
3. Linked Lists
4. Strings

Scalar Built-in Data Types

- Basic building blocks for other structures:
  1. Integers (int)
  2. Floating-point numbers (float)
  3. Characters (char)

- Implicit type conversion allow these data types to be mixed in an expression.
- Sometimes `cast`*ing* is required to for an expression to evaluate correctly

```
((float) x) / N
```

Data Structures

There is a famous saying that

“Algorithms + Data Structures = Programs” (Wirth)

“For many applications, the choice of the proper data structure is the only major decision involving the implementation: once the choice is made, the necessary algorithms are simple.” (Sedgewick)

- Suppose we have a list of sorted data on which we have to perform the following operations:
  - Search for an item
  - Delete a specified item
  - Insert (add) a specified item

Data Structures

Example: Suppose we begin with the following list:

data:	345	358	490	501	513	555	561	701	724	797
location:	0	1	2	3	4	5	6	7	8	9

- What is a list?
  - A list is a data structure where data is represented linearly
  - Finite sequence of items from the same data type
  - If arrays are used, items are stored contiguously in the memory

List Implementation using an Array

Example: suppose we begin with the following list:

data:	345	358	490	501	513	555	561	701	724	797
location:	0	1	2	3	4	5	6	7	8	9

Now, delete item 358 from the above list

Q: What is the algorithm to delete an item?

Q: What is the cost of deleting an item?

data:	345	358	490	501	513	555	561	701	724	797
location:	0	1	2	3	4	5	6	7	8	9

Q: When we delete 358, what happens to that location?

Now, add item 498 onto the above list

Q: Where would that item go? 498

data:	345	358	490	501	513	555	561	701	724	797
location:	0	1	2	3	4	5	6	7	8	9

Q: What is the cost of inserting an item?

Conclusion:

Using a list representation of data, what is the overall efficiency of searching, adding, and deleting items?

Lecture 2

Deletion of an Element from a List

- Algorithm:

1. locate the element in the list (this involves searching)
2. delete the element
3. reorganize the list and index

Example:

```
   data:  345 358   490   501   513   555   561   701
724   797
   location:     0     1     2     3     4     5     6
7     8     9
```

Delete 358 from the above list:

1. Locate 358: if we use ‘linear search’, we’ll compare 358 with each element of the list starting from the location 0.

Insertion of an Element in List

- Algorithm:
  1. locate the position where the element in to be inserted (position may be user-specified in case of an unsorted list or may be decided by search for a sorted list)
  2. reorganize the list and create an ‘empty’ slot
  3. insert the element

  Example: (sorted list)

  data: 345 358 490 501 513 555 561 701 724 797

  location: 0 1 2 3 4 5 6 7 8 9

  Insert 505 onto the above list:

  1. Locate the appropriate position by performing a binary search. 505 should be stored in location 4.
  2. Create an ‘empty’ slot

  data: 345 358 490 501 513 555 561 701 724 797

  location: 0 1 2 3 4 5 6

  3. Insert 505

Methods for defining a collection of objects

- Array
  - – successive items locate a fixed distance
- disadvantage
  - – data movements during insertion and deletion
  - – waste space in storing n ordered lists of varying size
- possible solution
  - – linked list
  - – linked lists are dynamically allocated and make extensive use of pointers

Lecture 3-4

Sorted Arrays

- a[i] is 'less than or equal to' a[i+1] for i = left..right-1
- Meaning of 'less than or equal to' can vary
- need a method of testing whether or not 'less than or equal to' is true

Linear Search in Sorted Array

Search for target in a[left..right]
This is an O(n) algorithm.

```
1. Loop using p = left..right
      1.1 If a[p] greater than or equal to target then exit loop

2. If a[p] equals target then return index p else return 'not
   found'
```

Binary Search (Recursive)

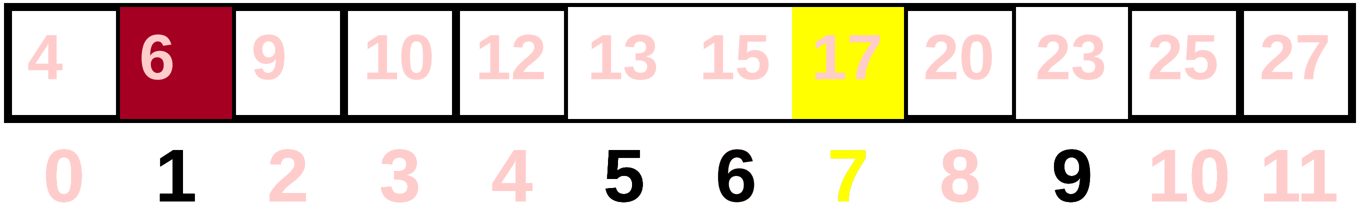
Search(target,a,left,right)

This is an O(log n) algorithm.

```
1. If left >= right then
     1.1 If a[left] equals target then
          1.1.1 return index left
     1.2 Else
          1.2.1 return -1 (i.e. 'not found’)
2. Set mid = (left+right)/2
3. If target is greater than a[mid] then
     3.1 return Search(target,a,mid+1,right)
4. Else
     4.1 return Search(target,a,left,mid)
```

Binary Search Example

Find position of integer 17 between indices 1 and 9



4	6	9	10	12	13	15	17	20	23	25	27
0	1	2	3	4	5	6	7	8	9	10	11

Binary Searching

The values to be searched must be sorted in order

Go to the mid point of the list or array

Compare this with the value to be found

If the value to be found is less than the mid point search the first half of the list or array

If the value to be found is greater than the mid point search the second half of the list or array

Divide the next part of the list or array in exactly the same way and perform the same comparisons until the item is found or no more searches can be made.

Find 14 Using Binary Search

positions

0	1	2	3	4	5	6	7	8
12	**14**	**25**	**39**	**41**	**56**	**78**	**88**	**90**

values

lowest = 0, highest = 8

Mid point (lowest+highest) / 2 = 4 value is 41

14 is less than 41

lowest = 0, highest = (mid -1) = 3 value is 39

Therefore search: **12** **14** **25** **39**

Mid point (lowest+highest) / 2 = 1 value is 14

Search value 14 = 14.

Find 88 Using Binary Search

12 14 25 39 41 56 78 88 90

lowest = 0, highest = 8

Mid point (lowest+highest) / 2 = 4 value is 41 88 is greater than 41

lowest = (mid + 1) = 5, highest = 8

Therefore search: **56 78 88 90**

Mid point (lowest+highest) / 2 = 6 value is 78 88 is greater than 78

lowest = (mid + 1)=7, highest = 8

Therefore search: **88 90**

Mid point (lowest+ highest) / 2 = 7 value is 88

Search value 88 = 88.

Introduction to Sorting [1/3]

To **sort** a collection of data is to place it in order.

We will deal primarily with algorithms that solve the General Sorting

In this problem, we are given:

- ☐ A sequence.
  - ☐ Items are all of the same type.
  - ☐ There are no other restrictions on the items in the sequence.
- ☐ A comparison function.
  - ☐ Given two sequence items, determine which should come first.
  - ☐ Using this function is the **only** way we can make such a determination.
- ☐ We return:
  - ☐ A sorted sequence with the same items as the original sequence.

Review: Introduction to Sorting [2/3]

We will analyze sorting algorithms according to five criteria:

- **Efficiency**
  - What is the (worst-case) order of the algorithm?
  - Is the algorithm much faster on average than its worst-case performance?
- **Requirements on Data**
  - Does the algorithm need random-access data? Does it work well with Linked Lists?
  - What operations does the algorithm require the data to have?
    - Of course, we always need “compare”. What else?
- **Space Usage**
  - Can the algorithm sort in-place?
    - An **in-place** algorithm is one that does not require extra buffers to hold a large number of data items.
  - How much additional storage is used?

Review: Introduction to Sorting [3/3]

There is no **known** sorting algorithm that has **all** the properties we would like one to have.

We will examine a number of sorting algorithms. Generally, these fall into two categories: $O(n^2)$ and $O(n \log n)$.

- Quadratic [$O(n^2)$] Algorithms
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Quicksort

Review:

Merge Sort does essentially everything we would like a sorting algorithm to do:

- It runs in $O(n \log n)$ time.
- It is stable.
- It works well with various data structures (especially linked lists).

Thus, Merge Sort is a good standard by which to judge sorting algorithms.

When considering some other sorting algorithm, ask

The Importance Of Algorithm Analysis

- Performance matters! Can observe and/or analyze, then tune or revise algorithm.
- Algorithm analysis is SOOOO important that every Brown CS student is *required* to take at least one course in it!
- CS16: Introduction to Algorithms and

Analysis of Algorithms

- Computing *resources* consumed
  - running time
  - memory space

- *Implementation* of algorithm
  - machine (Intel Core 2 Duo, AMD Athlon 64 X2,...)
  - language (Java, C++,...)

Big-O Notation - OrderOf()

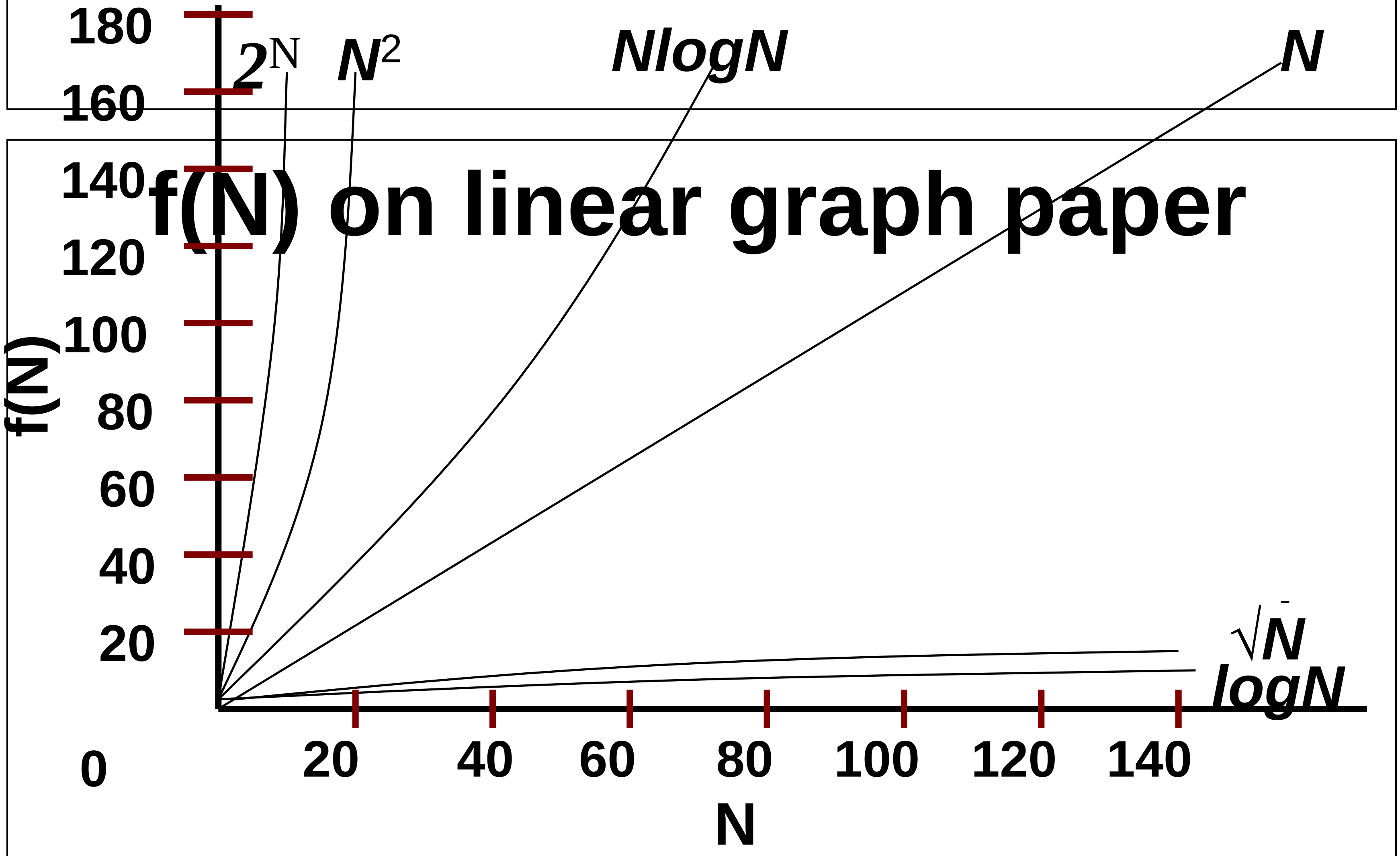
- How to *abstract* from implementation?

This image cannot currently be displayed.

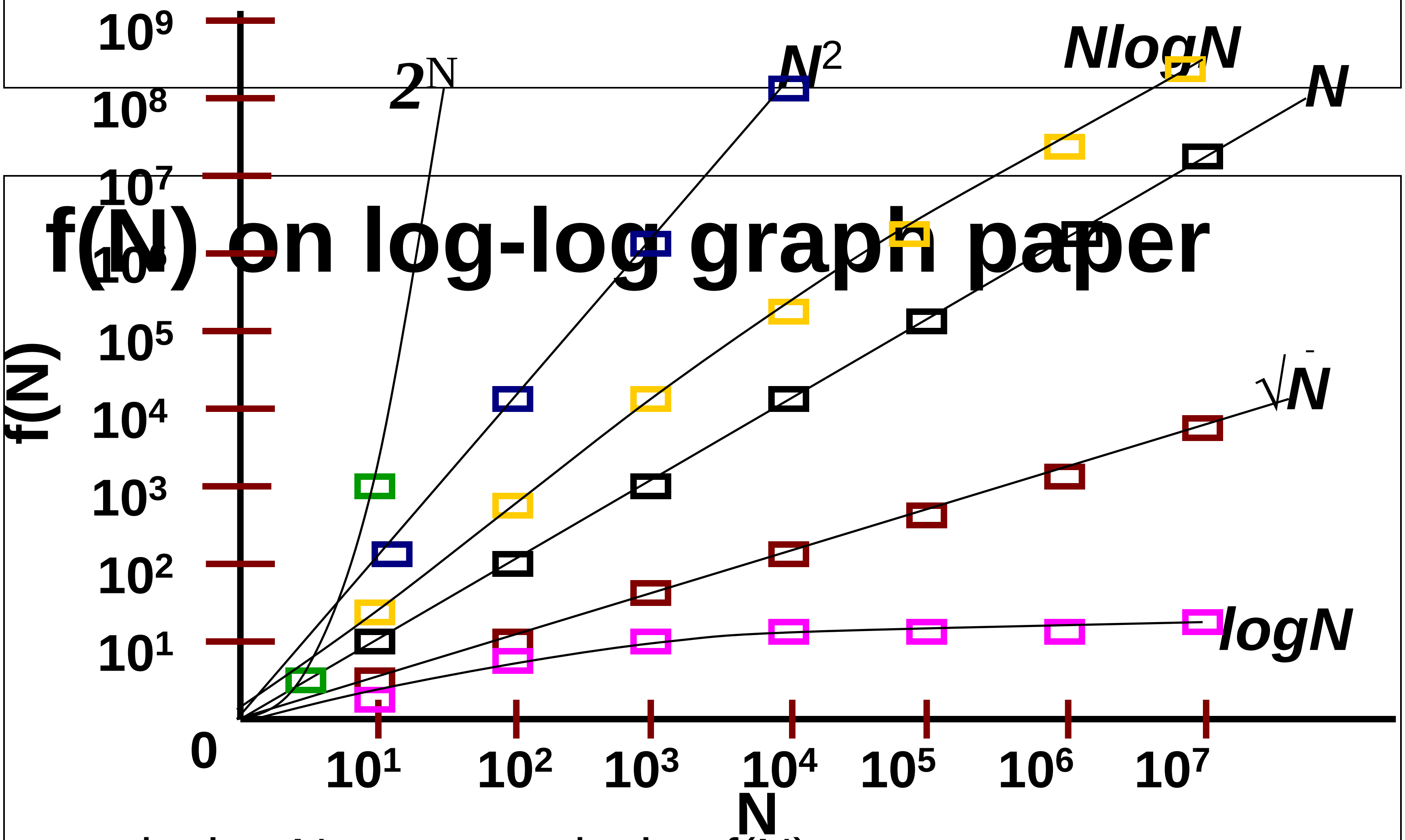
- *Big-O* notation
- *O(N)* means each element is accessed once
  - – N elements * 1 access/element = N accesses
- *$O(N^2)$* means each element is accessed n times
  - – N elements * N accesses/element = N^2 accesses

$$\sum_{i=1}^{n} i = N(N+1)/2 \text{ is } O(N^2)$$

f(N) on linear graph paper



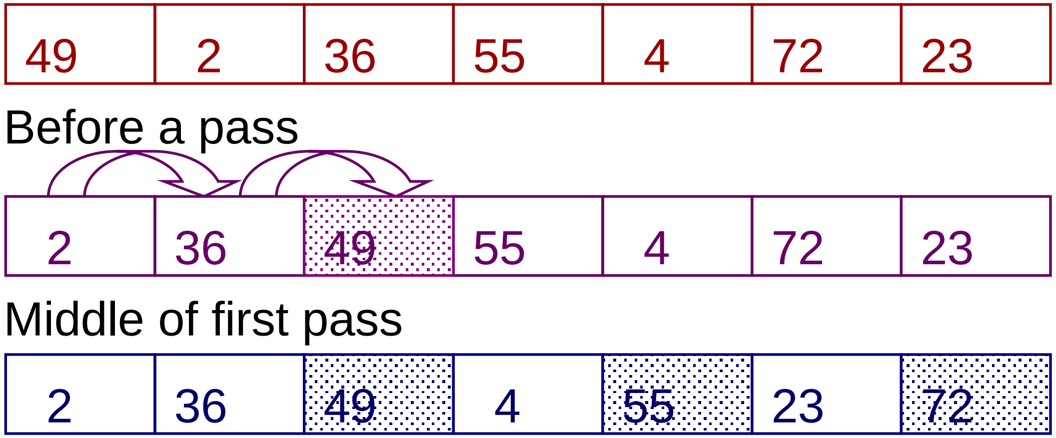
f(N) on log-log graph paper



- x-axis: log N; y-axis: log f(N)
- the diagram of cf(N) is obtained by “shifting” the diagram of f(N) up by log c.

Bubble Sort

- Iterate through sequence, compare each element to right neighbor.
- Exchange adjacent elements if necessary.
- Keep passing through sequence until no exchanges are required (up to N times).
- Each pass causes largest element to bubble into place: 1st pass, largest; 2nd pass, 2nd largest, ...



49	2	36	55	4	72	23

Before a pass

2	36	49	55	4	72	23

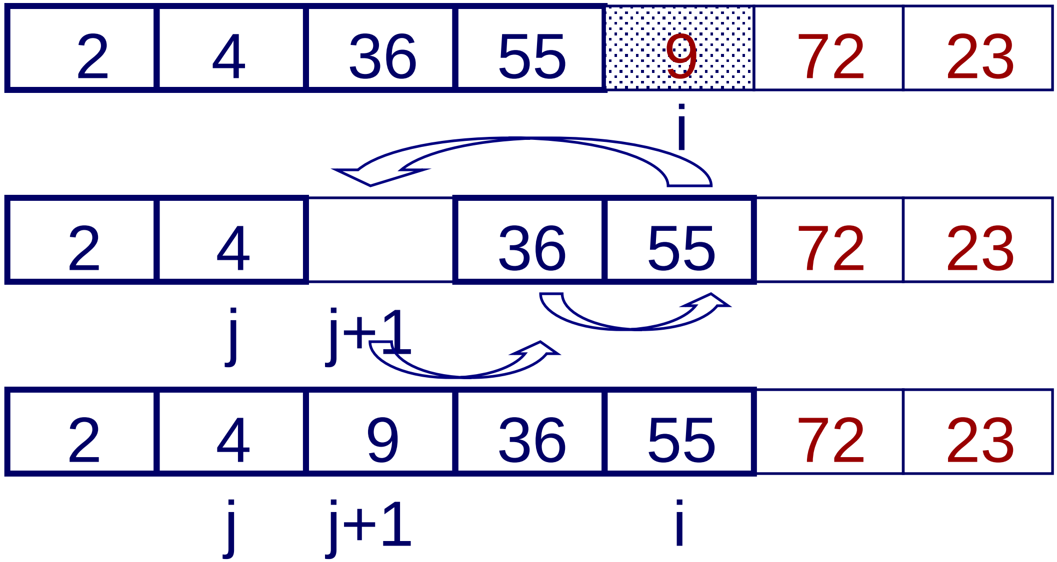
Middle of first pass

2	36	49	4	55	23	72

After one pass

- Like inserting new card into a partially sorted hand by bubbling to left into sorted subarray; little less brute-force than bubble sort
  - add one element a[i] at a time
  - find proper position, j+1, to the left by shifting to the right a[i-1], a[i-2], ..., a[j+1] left neighbors, til a[j] < a[i]
  - move a[i] into vacated a[j+1]
- After iteration i<n, a[1] ... a[i] are in sorted order, but not necessarily in final position

Insertion Sort



Time Complexity of Insertion Sort

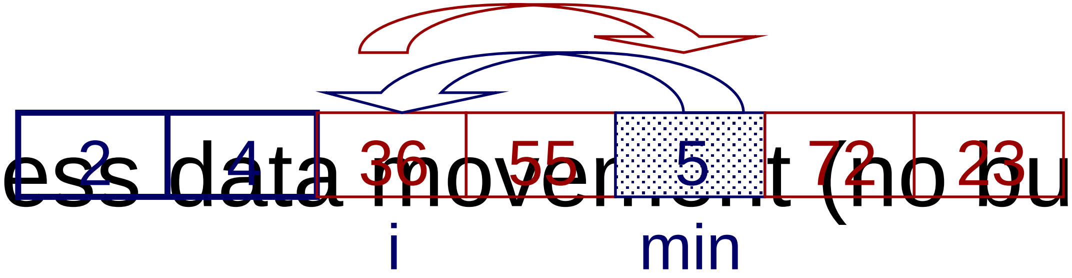
Pseudocode implementation

```
for (int i = 2; i <= n; i++) {

    int j;
    for (j = i - 1; (j > 0) &&
                (a[j] > a[i]); j--) {
      move a[j] forward;
    }

    move a[i] to a[j+1];
}
```

Lecture 4

Selection Sort

- Find smallest element and put it in a[1].
- Find 2nd smallest element and put it in a[2].
- etc. a Less data movement (no bubbling)



Pseudocode:

Time Complexity of Selection Sort

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

Comparison of Elementary Sorting Algorithms

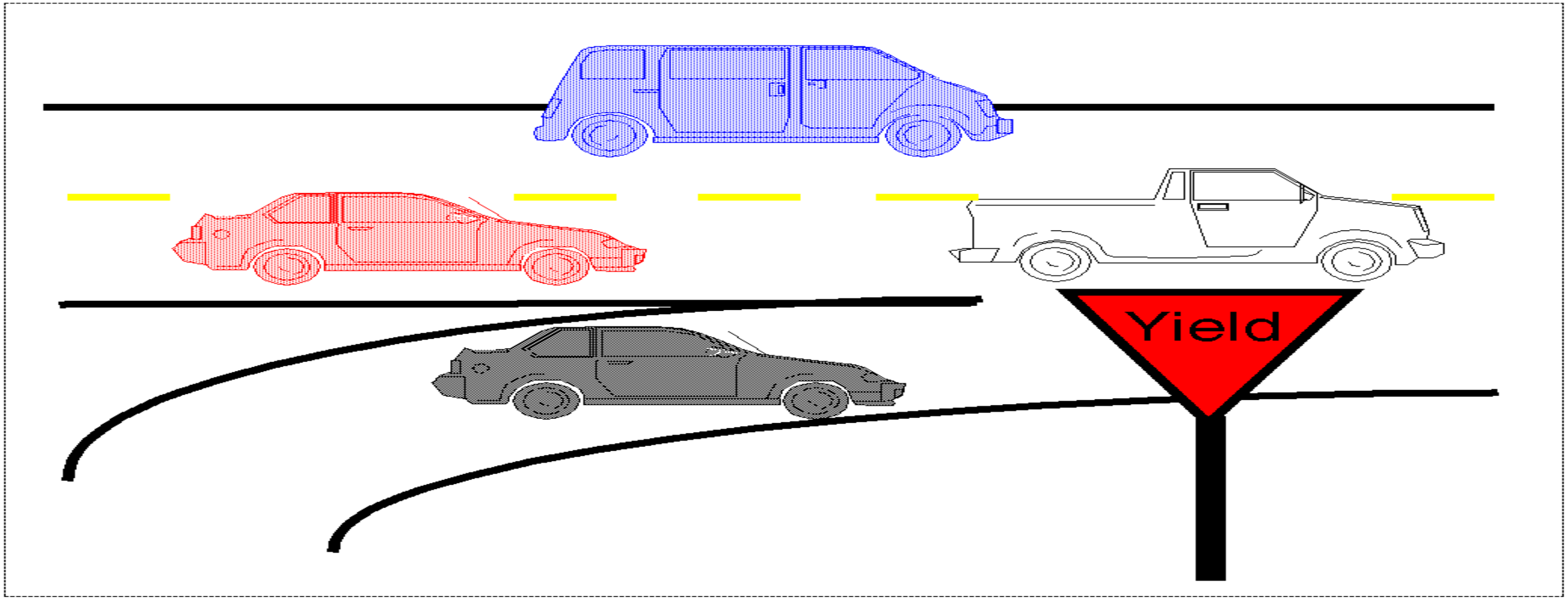
Note: smaller terms omitted

		Selection	Insertion	Bubble
Comparisons	**Best**	$\frac{n^2}{2}$	n	n
	Average	$\frac{n^2}{2}$	$\frac{n^2}{4}$	$\frac{n^2}{2}$
	Worst	$\frac{n^2}{2}$	$\frac{n^2}{2}$	$\frac{n^2}{2}$
Movements	**Best**	0	0	0
	Average	n	$\frac{n^2}{4}$	$\frac{n^2}{2}$
	Worst	n	$\frac{n^2}{2}$	$\frac{n^2}{2}$

Lecture 5-6

Merge Sort

- *Divide-and-Conquer* algorithm
- Time complexity: *$O(N \log N)$*



keys

Merging Two Sorted Lists

A	1	5	9	25	**M**

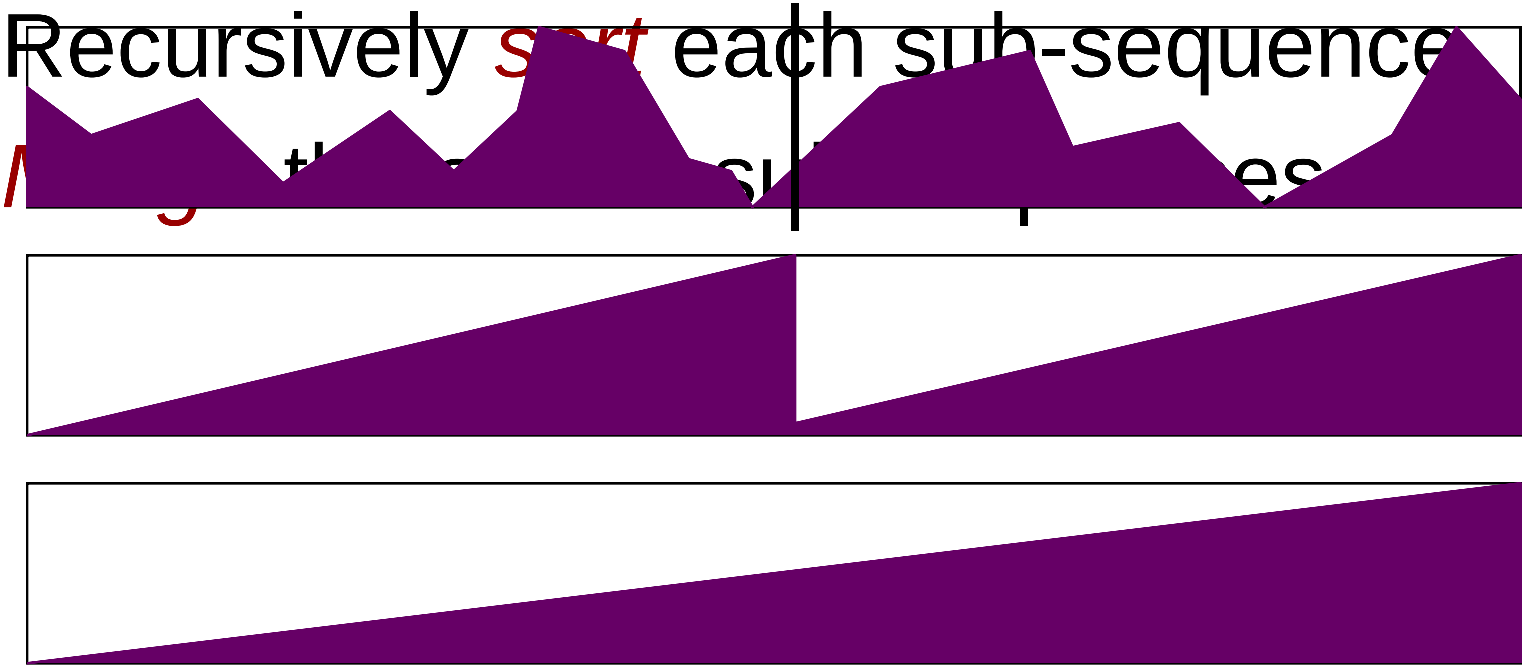
B	2	3	17	**N**

M + N

C	1	2	3	5	9	17	25

Outline of Recursive (Top Down) Merge Sort

- *Partition* sequence into two sub-sequences of N/2 elements.
- Recursively *sort* each sub-sequence.
- *M*[illegible] th[illegible] s[illegible] es.

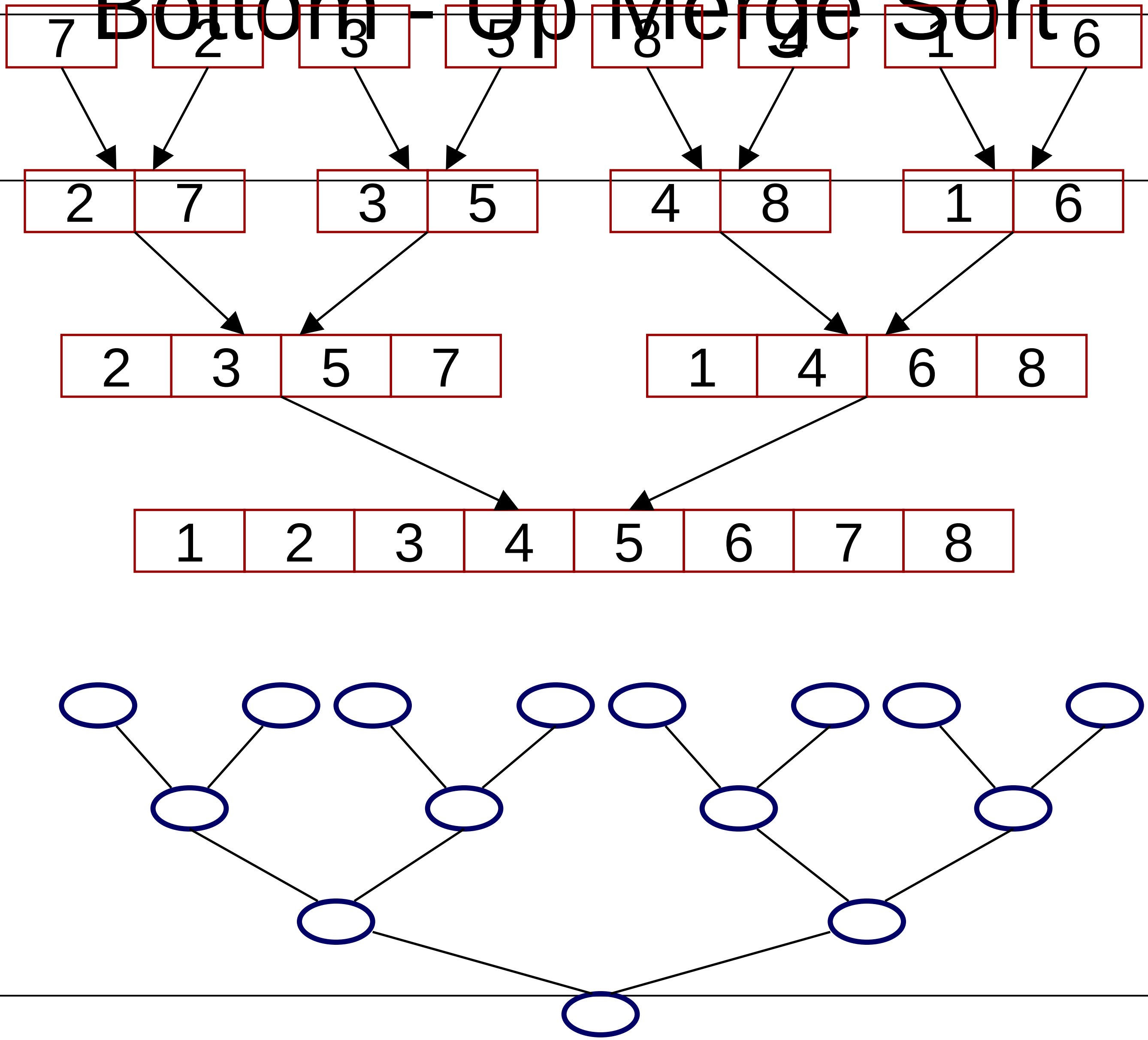


Recursive Merge Sort

- `listSequence` is sequence to sort.
- `first` and `last` are smallest and largest indices of sequence.

```
public class Sorts {
  // other code here
  public void mergeSort(
          ItemSequence listSequence;
          int first, int last) {
      if (first < last) {
```

Bottom - Up Merge Sort



7 2 3 5 8 4 1 6

2 7 3 5 4 8 1 6

2 3 5 7 1 4 6 8

1 2 3 4 5 6 7 8

Bottom - Up Merge Sort

```
for k = 1, 2, 4, 8, ... , N/2 {
   merge all pairs of consecutive
   sub-sequences of size k into sorted
   sub-sequences of size 2k.
}
```

- Number of *iterations* is *$\log_2 N$*

Time Complexity of Recursive Merge Sort

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N \qquad \text{for } N \geq 2$$

merge

2 recursive sorts

$$T(1) = 1$$

(1) $T(N) = 2 \cdot T\left(\frac{N}{2}\right) + N$

$T(N) \overset{?}{=} 2 \cdot \left[2T\left(\frac{N}{4}\right) + \frac{N}{2} \right] + N$

(2) $T(N) \overset{?}{=} 4 \cdot T\left(\frac{N}{4}\right) + 2 \cdot \frac{N}{2} + N$

(3) $T(N) \overset{?}{=} 8 \cdot T\left(\frac{N}{8}\right) + 4 \cdot \frac{N}{4} + N$

$\vdots$

(i) $2^i T\left(\frac{N}{2^i}\right) + \overbrace{N + N + \ldots + N}^{i}$

Lecture 7

Quicksort — Introduction

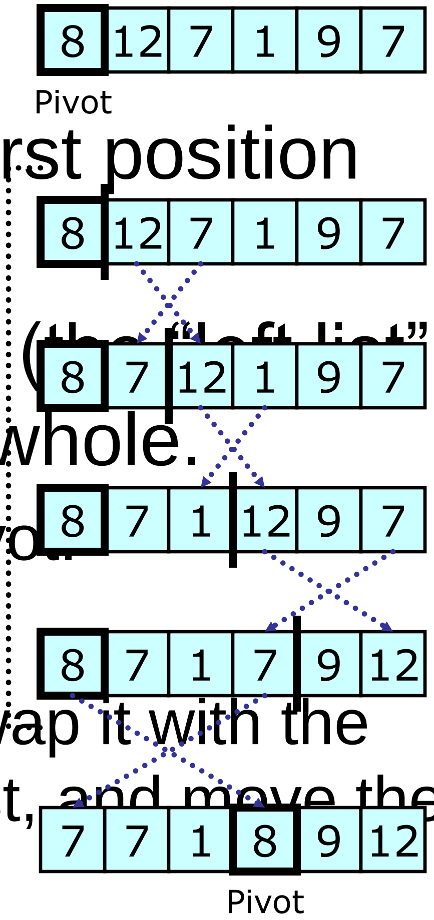
The divide-and-conquer idea used in Binary Search and Merge Sort is a good way to get fast algorithms.

What about this variation:

- Pick an item in the list.
  - This first item will do — for now.
  - This item is the **pivot**.
- Rearrange the list so that the items before the pivot are all less than, or equivalent to, the pivot, and the items after the pivot are all

Review:
Sorting Algorithms III —

An In-Place Partition Algorithm



- Make sure the pivot lies in the first position (swap if not).
- Create a list of items ≤ the pivot (the “left list”) at the beginning of the list as a whole.
  - Start: the left list holds only the pivot.
  - Iterate through rest of the list.
  - If an item is less than the pivot, swap it with the item just past the end of the left list, and move the left-list end mark one to the right.
- Lastly, swap the pivot with the last item in the left list

Review:
Sorting Algorithms III —

Quicksort has a big problem.

- Try applying the Master Theorem. It doesn't work, because Quicksort may not split its input into nearly equal-sized parts.
- The pivot *might* be chosen very poorly. In such cases, Quicksort has linear recursion depth and does linear-time work at each step.
- Result: Quicksort is $O(n^2)$. ☹
- And the worst case happens when the data are **already sorted**!

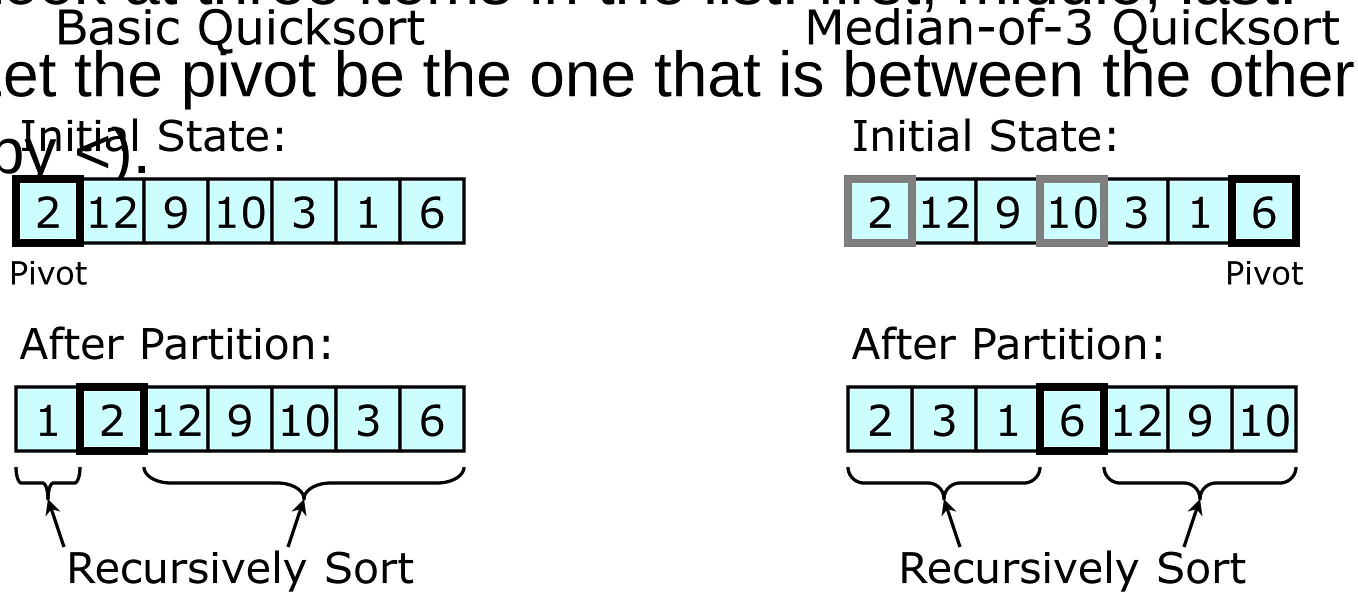
However, Quicksort's **average-case** time is very fast.

This is $O(n \log n)$ and significantly faster than Merge

Review: Sorting Algorithms III —

Choose the pivot using **median-of-three**.

- Look at three items in the list: first, middle, last.
- Let the pivot be the one that is between the other two (by <).



Basic Quicksort

Initial State:

2 | 12 | 9 | 10 | 3 | 1 | 6

Pivot

After Partition:

1 | 2 | 12 | 9 | 10 | 3 | 6

Recursively Sort

Median-of-3 Quicksort

Initial State:

2 | 12 | 9 | 10 | 3 | 1 | 6

Pivot

After Partition:

2 | 3 | 1 | 6 | 12 | 9 | 10

Recursively Sort

Sorting Algorithms III, cont'd: Quicksort — Improvements:

How much additional space does Quicksort use?

- Quicksort is in-place and uses few local variables.
- But it is recursive.
- Quicksort's additional space usage is thus proportional to its recursion depth.
- And that is linear. Additional space: *O*(*n*).

We can improve this:

- Do the **larger** of the two recursive calls last

Sorting Algorithms III, cont'd: Quicksort — Do It #2

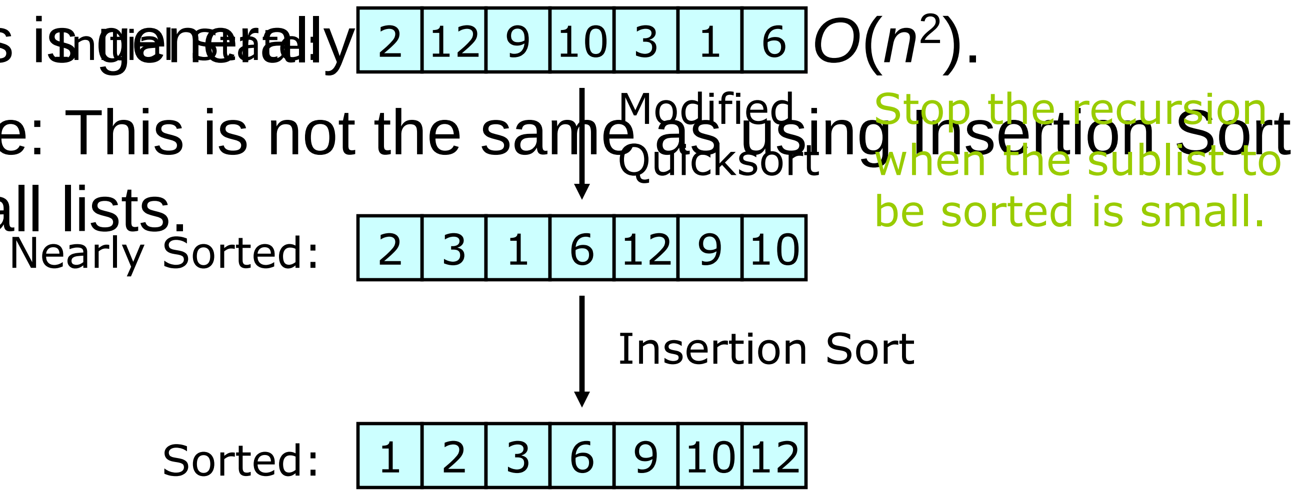
To Do

- Rewrite our Quicksort to do: (*Done. See* **`quicksort2.cpp`**, *on the web page.*)
  - Reduced recursion depth.
  - Median-of-three pivot selection.

Sorting Algorithms III, cont'd: Quicksort — Improvements:

A Minor Speed-Up: Finish with Insertion Sort

- Stop Quicksort from going to the bottom of its recursion. We end up with a nearly sorted list.
- Finish sorting this list using one call to Insertion Sort.
- This is generally $O(n^2)$.
- Note: This is not the same as using Insertion Sort for small lists.



Initial State: 2 | 12 | 9 | 10 | 3 | 1 | 6

Modified Quicksort

Stop the recursion when the sublist to be sorted is small.

Nearly Sorted: 2 | 3 | 1 | 6 | 12 | 9 | 10

Insertion Sort

Sorted: 1 | 2 | 3 | 6 | 9 | 10 | 12

Sorting Algorithms III, cont'd: Quicksort — Improvements:

We want an algorithm that:

- Is as fast as Quicksort on the average.
- Has reasonable [*O*(*n* log *n*)] worst-case performance.

But for over three decades no one found one.

Some said (and some still say) "Quicksort's bad behavior is very rare; ignore it."

- I suggest to you that this is not a good way to think.
- Sometimes bad worst-case behavior is okay; sometimes it is not. Know what is important in the situation you are addressing.
- From the Wikipedia article on Quicksort (retrieved 18 Oct 2006):

> The worst-case behavior of quicksort is not merely a theoretical problem. When quicksort is used in web services, for example, it is possible for an attacker to deliberately exploit the worst case performance and choose data which will cause a slow running time

Sorting Algorithms III, cont’d: Quicksort — Analysis

- Efficiency ☹
  - Quicksort is $O(n^2)$.
  - Quicksort has a **very** good $O(n \log n)$ average-case time. ☺☺
- Requirements on Data ☹
  - Non-trivial pivot-selection algorithms (median-of-three and others) are only efficient for random-access data.
- Space Usage 😐
  - Quicksort can be done efficiently in-place.
  - Quicksort uses space for recursion.
    - Additional space: $O(\log n)$, if you are clever about it.
    - Even if **all** recursion is eliminated, $O(\log n)$ additional space is used.
    - This additional space need not hold any data items.

Lecture 8

Radix Sort: Description

A practical algorithm can be based on this bucket sorting method.

Suppose we want to sort a list of **strings** (in some sense):

- Character strings.
- Numbers, considered as strings of digits.
- Short-ish sequences of some other kind.
- I will call the entries in a string “characters”.

We want to sort in **lexicographic order**.

- This means sort first by first character, then by second, etc.
- For strings of letters, this is alphabetical order.
- For positive numbers padded with leading zeroes, this

Radix Sort: Example

Start with the following list:

- 583, 508, 134, 183, 90, 223, 236, 924, 4, 426, 106, 624.

We first organize them by the units digit:

- <u>90</u>, <u>583, 183, 223</u>, <u>134, 924, 4, 624</u>, <u>236, 426, 106</u>, <u>508</u>.

Then we do it again, based on the tens digit, not reversing the order of items with the same tens digit:

- 4, 106, 508, 223, 924, 624, 426, 134, 236, 583, 183, 90

Radix Sort:
Do It #2

To Do

- Write Radix Sort for small-ish positive integers. *Done. See* **`radix_sort.cpp`**, *on the web page.*

Radix Sort: Efficiency [1/2]

How Fast is Radix Sort?

- Fix the number of characters and the character set.
- Then each sorting pass can be done in linear time.
  - Use the bucket method.
  - Create a bucket for each possible character.
- And there are a fixed number of passes.
- Thus, Radix Sort is *O*(*n*): **linear time**.

How is this possible?

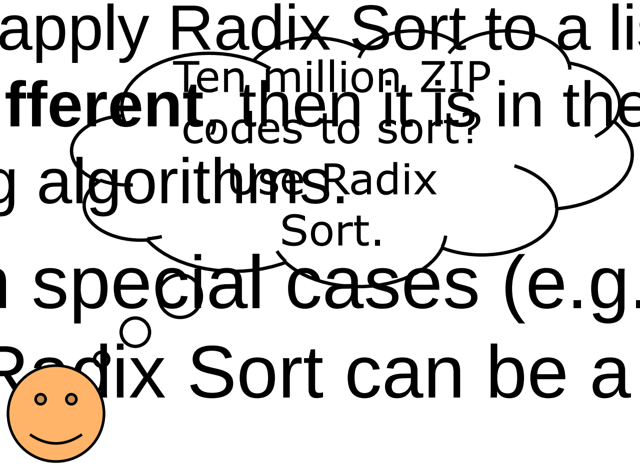
- Radix Sort places a list of values in order.
- However, it does *not* solve the General Sorting Problem.
  - information about our data is by applying a given comparison

Radix Sort: Efficiency [2/2]

Radix Sort is not as efficient as it might seem.

- There is a hidden logarithm. The number of passes required is equal to the length of a string, which is something like the logarithm of the number of possible values.
- So if we want to apply Radix Sort to a list in which all the values are **different**, then it is in the same class as normal sorting algorithms.

However, in certain special cases (e.g., big lists of small numbers) Radix Sort can be a very effective tool.



Lecture 9

Shell Sort

- Also called Diminishing Increment sort. Invented by Donald Shell in 1959.
- Another refinement of the Straight Insertion sort.
- In each step, sort every *kth* item. Then sort the **sublists**,

  a[0], a[k], a[2k], a[3k], etc.

  a[1], a[k+1], a[2k+1], a[3k+1], etc.

  … …

  a[k-1], a[2k-1], a[3k-1], a[4k-1], etc.
- After these sublists are sorted, chose a new, smaller value for k, and sort the new sublists.
- Finally, sort with k = 1. This is the Insertion sort!

Shell Sort, example

44 55 12 42 94 18 06 67

- 4-sort yields:

44 18 06 42 94 55 12 67

- 2-sort yields

06 18 12 42 44 55 94 67

- 1-sort yields

Shell Sort, example 2

44 55 12 42 94 18 06 67

- 5-sort yields:

18 06 12 42 94 44 55 67

- 3-sort yields

18 06 12 42 67 44 55 94

- 1-sort yields

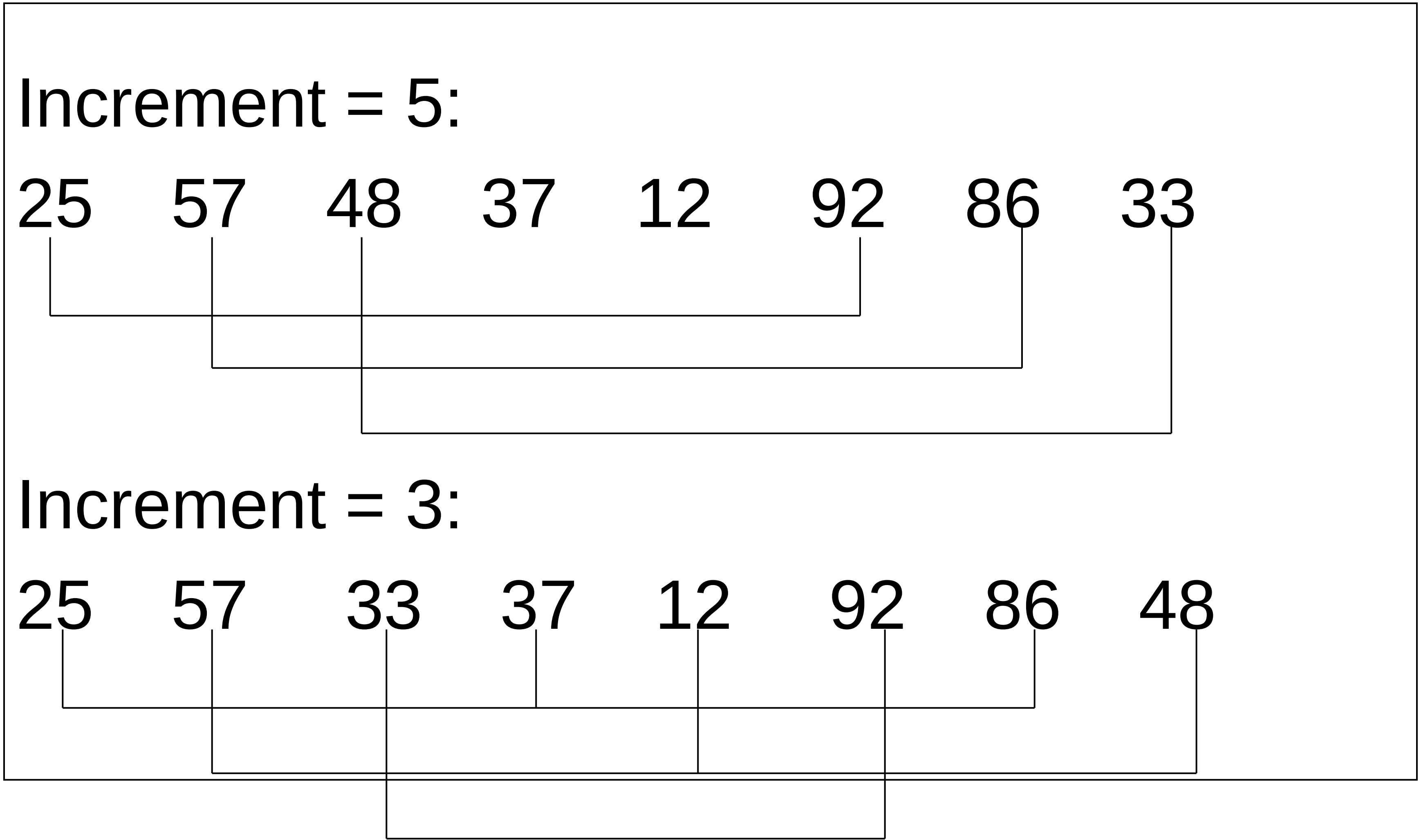
06 12 18 42 44 55 67 94

Shell Sort

- Each pass benefits from the previous passes.
- Each pass partially sorts a relatively small portion of the full list. Since the sublists are fairly small, insertion sort is efficient for sorting the sublists.
- As successive passes use smaller increments (and thus larger sublists), they are almost sorted due to previous passes.

Each partial sort DOES NOT DISTURB

Shell Sort - example



Increment = 5:

25 57 48 37 12 92 86 33

Increment = 3:

25 57 33 37 12 92 86 48

Example, cont.

Increment = 1:

25 12 33 37 48 92 86 57

Final Result:

12 25 33 37 48 57 86 92

Shell Sort

- How are the increments chosen? Any sequence will work, as long as the last pass has k = 1.
- Analysis is complicated. It has been demonstrated that:
  - Increments should be relatively prime (i.e., share no common factors). This guarantees that successive passes intermingle sublists so that the entire list is almost sorted before the final pass.
  - With well-chosen increments, efficiency can

Shell Sort - cont.

- Many text books (e.g., Knuth) describe a good algorithm for determining increments.
  - Set K = to the number of items in the list
  - Set K = K **/** 3 + 1

Shell Sort, cont.

```
increment = count;
do {
    increment = increment / 3 + 1;
    for (start = 0; start < increment; start++)
        “sort sub list (start, increment, count);”
while (increment > 1);
```

Shell Sort, review

- **ALGORITHM**: Another refinement of the Insertion sort. In each step, sort every *kth* item. Then sort the sublists,

  a[0], a[k], a[2k], a[3k], etc.
  a[1], a[k+1], a[2k+1], a[3k+1], etc.
  a[2], a[k+2], a[2k+2], a[3k+2], etc.
  … …
  a[k-1], a[2k-1], a[3k-1], a[4k-1], etc.

  After these sublists are sorted, chose a new, smaller value for k

  and sort the new sublists. Finally, sort with k = 1.
- **PERFORMANCE**: **$O(n*\log(n)^2)$**
- **SPACE REQUIREMENTS and COMMENTS**: Need space for original list plus one additional temporary location.

  Good method for moderately sized N.

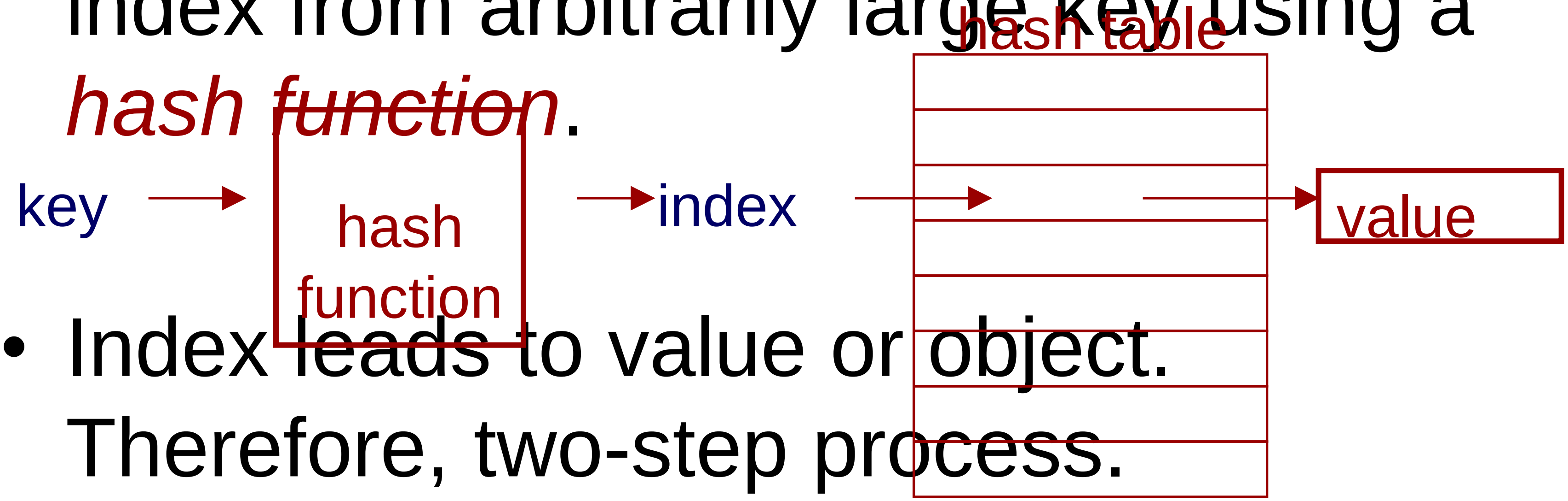
Shell Sort Review, cont.

- Analysis is complicated. Still do NOT know the best increments(!). But
  - Increments should be relatively prime (i.e., share no common factors; that is, they should not be multiples of each other). This guarantees that successive passes intermingle sublists so that the entire list is almost sorted before the final pass.
  - A good algorithm (taken from Knuth) for determining increments:
    - Set K = to the number of items in the list
    - Set K = K / 2 + 1

Lecture 10

Introduction to Hashing

- *Hashing* refers to deriving sequence index from arbitrarily large key using a *hash function*.



key

hash function

index

hash table

value

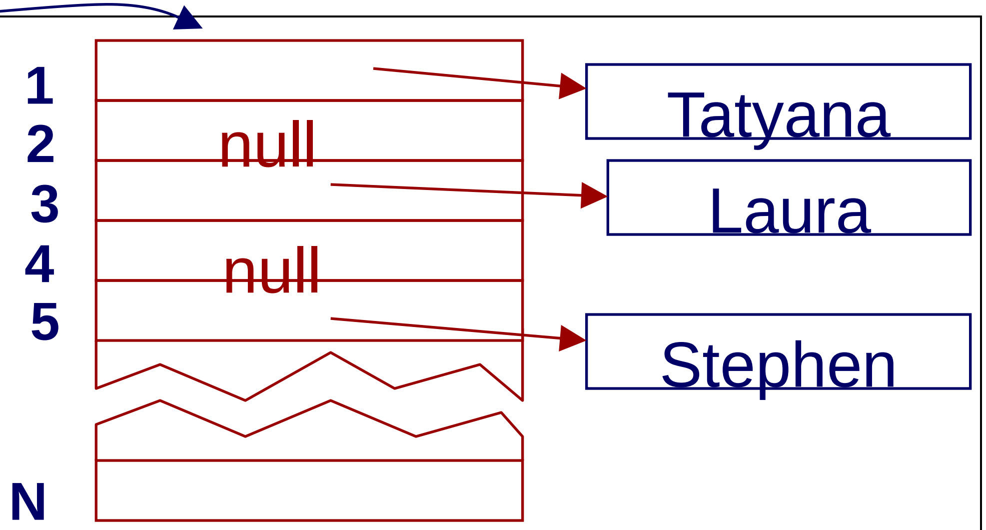
- Index leads to value or object. Therefore, two-step process.

Introduction to Hashing (cont.)

Hash('Tatyana')=1

Hash('Laura')=3

Hash('Stephen')=5


sequence of links to instances of the class **TA**

- Hash table typically holds several

Collisions

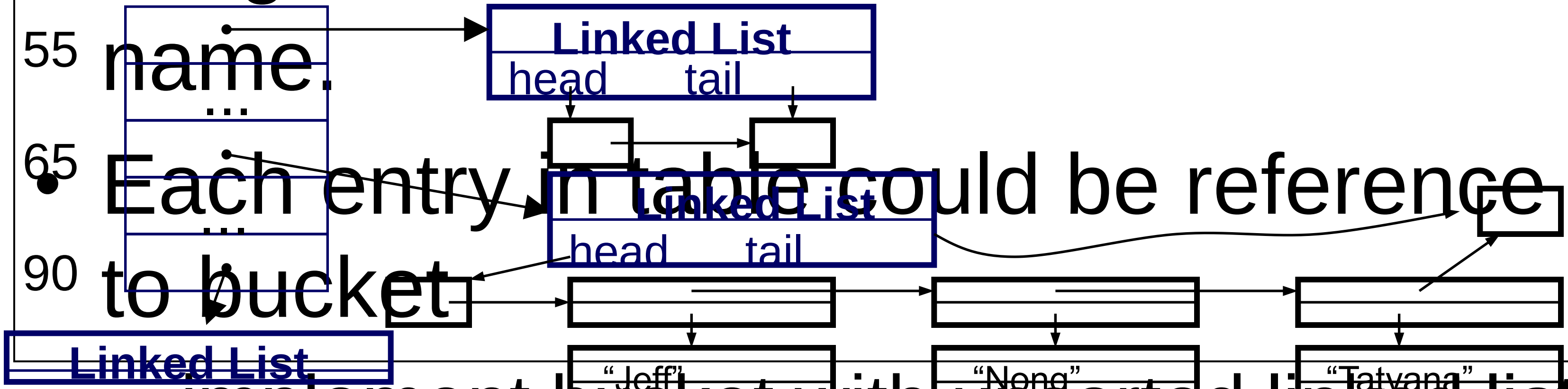
- Problem: Normally have more keys than entries in our table. Therefore inevitable that two keys hash to same position…
  - e.g., Hash(‘Sam’) = 4
  - and, Hash(‘Andrew’) = 4

- Called *collision* – multiple values hashed to the same key

Handling Collisions

- Since by design, we can't avoid collisions, we use *buckets* to catch extra entries.
- Consider stupid hash that returns integer value of first letter of each name.
- Each entry in table could be reference to bucket
  - implement bucket with unsorted linked list

    then we only need to search within (small)



Building a Good Hash Function

- Good hash functions
  - – take into account *all* information in key
  - – fill out hash table as *uniformly* as possible

- Thus, function that uses only first character (or any character) is *terrible* hashing function.
  - – Not many Q’s or Z’s, lots of A’s, M’s, etc

% (*mod*) provides simple method of

UNIT 3

Linked List

Lecture 1

Linked Lists

- Definition: a list of items, called **nodes**, in which the order of the nodes is determined by the address, called the **link**, stored in each node.

- Every node in a linked list has two components: one to store the relevant information (the data); and one to store the address, called the **link**, of the next node in the list.

Linked Lists

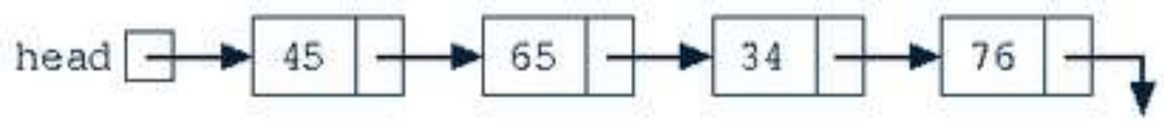
- The address of the first node in the list is stored in a separate location, called the **head** or **first**.

- The data type of each node depends on the specific application—that is, what kind of data is being processed; however, the link component of each node is a pointer. The data type of this pointer variable is the node type itself.

Linked Lists

Structure of a node

Structure of a linked list



Linked Lists: Some Properties

- The address of the first node in a linked list is stored in the pointer head
- Each node has two components: one to store the info; and one to store the address of the next node
- head should always point to the first node

Linked Lists: Some Properties

- Linked list basic operations:
  - Search the list to determine whether a particular item is in the list
  - Insert an item in the list
  - Delete an item from the list

Linked Lists: Some Properties

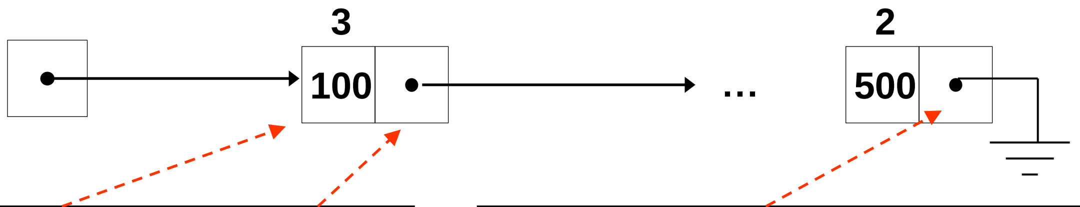
- These operations require traversal of the list. Given a pointer to the first node of the list, step through each of the nodes of the list
- Traverse a list using a pointer of the same type as head

List Implementation using Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Link pointer in the last node is set to null to mark the list's end
- Use a linked list instead of an array when
  - You have an unpredictable number of data elements

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **`NULL`** pointer (`0`)
- Diagram of two self-referential structure objects linked together



Data member and pointer

NULL pointer (points to nothing)

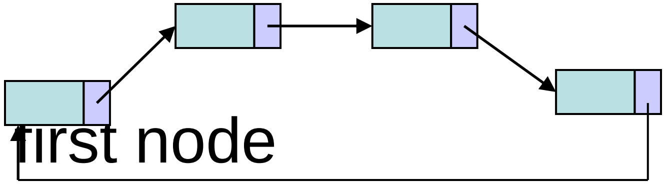
```
struct node {
        int data;
        struct node *nextPtr;
```

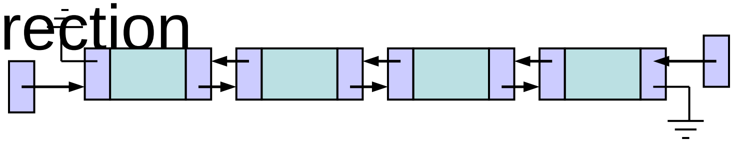
`nextPtr`

Lecture 2

Linked Lists

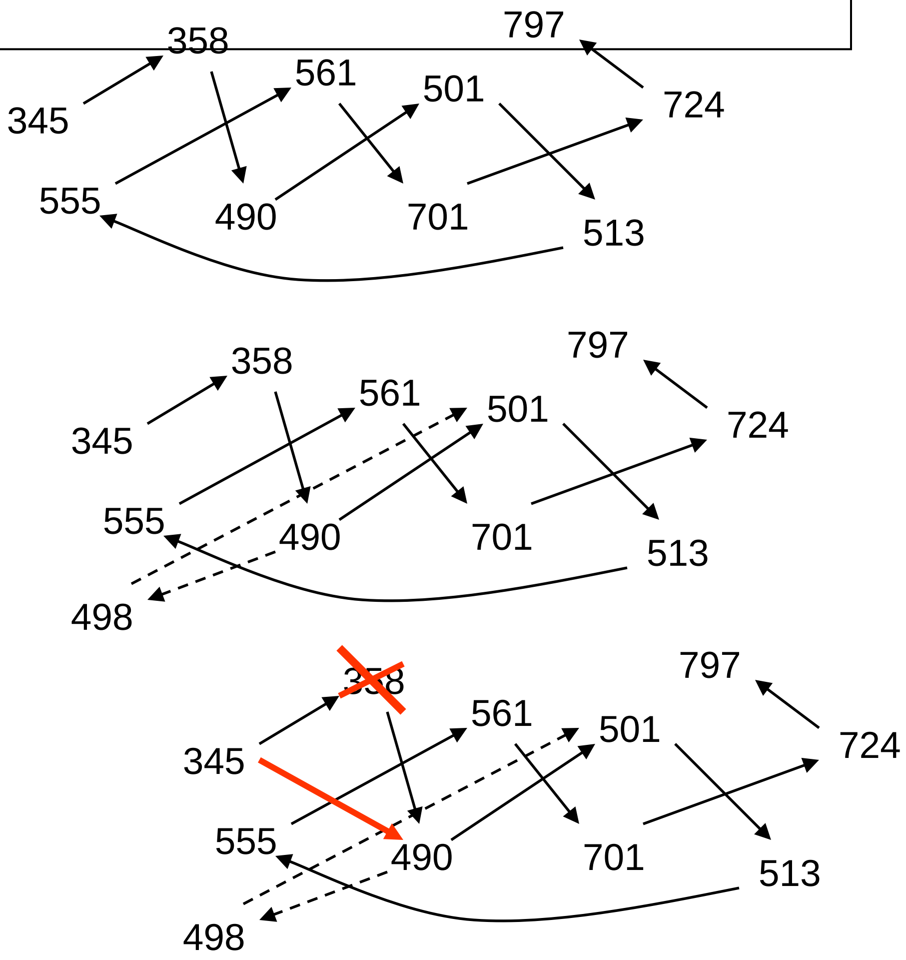
- Types of linked lists:
  - Singly linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - Circular, singly linked
    - Pointer in the last node points back to the first node
  - Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer





Linked Representation of Data

- In a linked representation, data is not stored in a contiguous manner. Instead, data is stored at random locations and the current data location provides the information regarding the location of the next data.



Adding item **498** on to the linked list

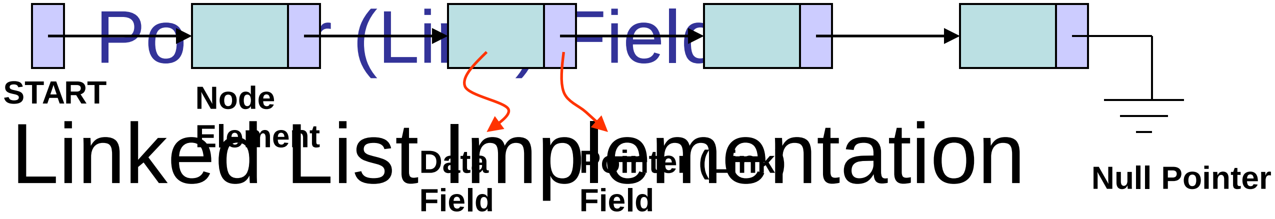
Q: What is the cost of adding an item?

Q: how about adding **300** and

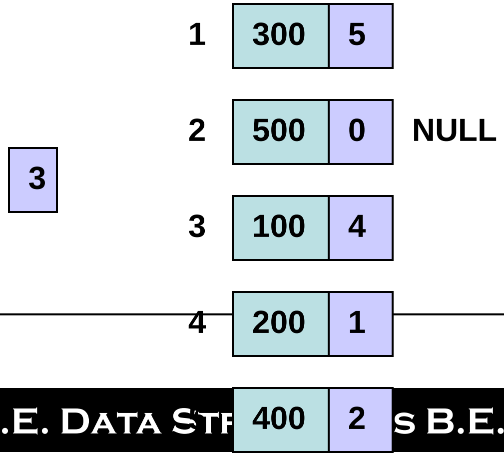
onto the linked list

Linked List

- How do we represent a linked list in the memory
  - Each location has two fields: Data Field and Po r (Li) Fiel



- Linked List Implementation



1	300	5
2	500	0 NULL
3	100	4
4	200	1
	400	2

-
```
struct node {
    int data;
    struct node *link;
 };
```

Conventions of Linked List

There are several conventions for the link to indicate the end of the list.

1. a *null link* that points to no node (0 or NULL)
2. a *dummy node* that contains no item
3. a reference back to the first node, making it a *circular list*.

Lecture 3

Singly Linked List

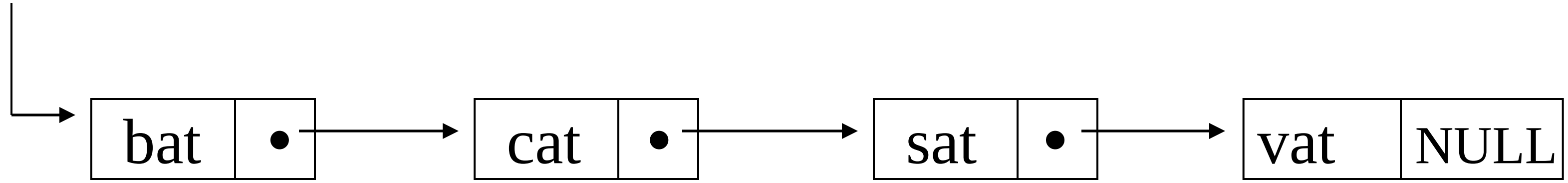
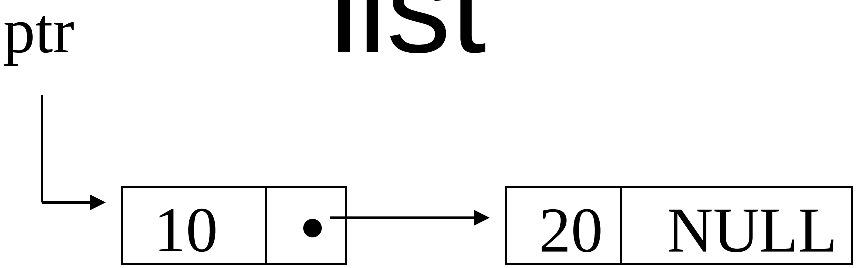


***Figure 4.1:** Usual way to draw a linked list (p.139)

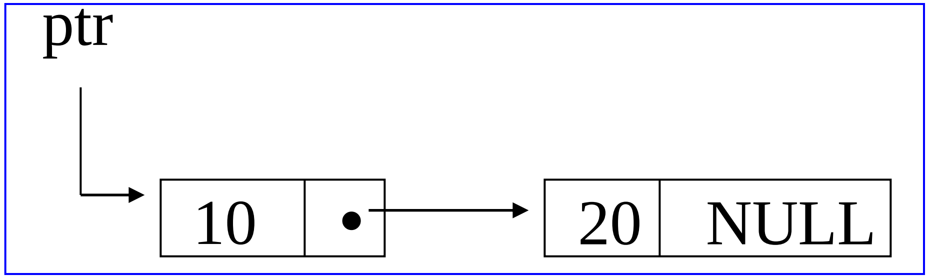
Example: create a two-node list



```
typedef struct list_node *list_pointer;
typedef struct list_node {
            int data;
            list_pointer link;
            };
list_pointer ptr =NULL
```

Two Node Linked List

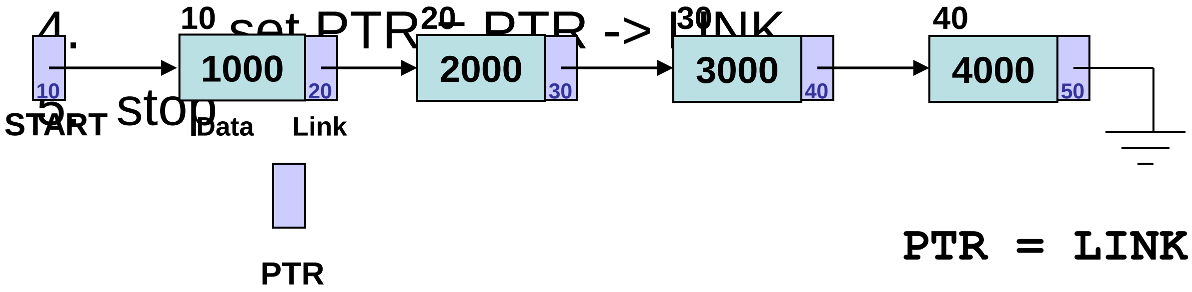
```
list_pointer create2( )
{
/* create a linked list with two nodes */
    list_pointer first, second;
    first = (list_pointer) malloc(sizeof(list_node));
    second = ( list_pointer) malloc(sizeof(list_node));
    second -> link = NULL;
    second -> data = 20;
    first -> data = 10;
    first ->link = second;
    return first;
}
```



Linked List Manipulation Algorithms

- List Traversal
  - Let START be a pointer to a linked list in memory. Write an algorithm to print the contents of each node of the list
  - Algorithm

```
1. set  PTR = START
2. repeat step 3 and 4 while PTR ≠ NULL
3.        print PTR->DATA
4.        set PTR = PTR -> LINK
5. stop
```

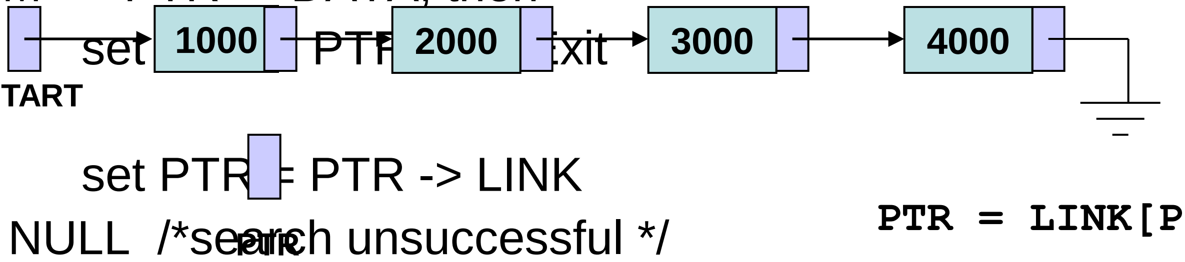


PTR = LINK[PTR]

Search for an Item

- Search for an ITEM
  - Let START be a pointer to a linked list in memory. Write an algorithm that finds the location LOC of the node where ITEM first appears in the list, or sets LOC=NULL if search is unsuccessful.
  - Algorithm

```
1.  set  PTR = START
2.  repeat step 3 while PTR ≠ NULL
3.        if ITEM == PTR -> DATA, then
4.              set          PTR         exit
5.        else
6.              set PTR = PTR -> LINK
7.  set LOC = NULL  /*search unsuccessful */
8.  Stop
```



PTR = LINK[PTR]

Lecture 4

Insertion

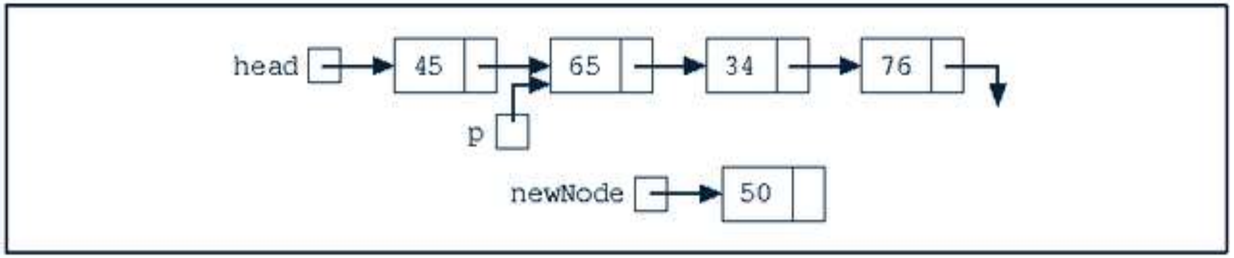


Figure 5-8 Create newNode and store 50 in it

- A linked list with pointers p and q

- newNode needs to be inserted

Insertion

- Code Sequence I
  - newNode→link = q
  - p→link = newNode

- Code Sequence II
  - p→link = newNode
  - newNode→link = q

Insertion

- Both code sequences produce the result shown below

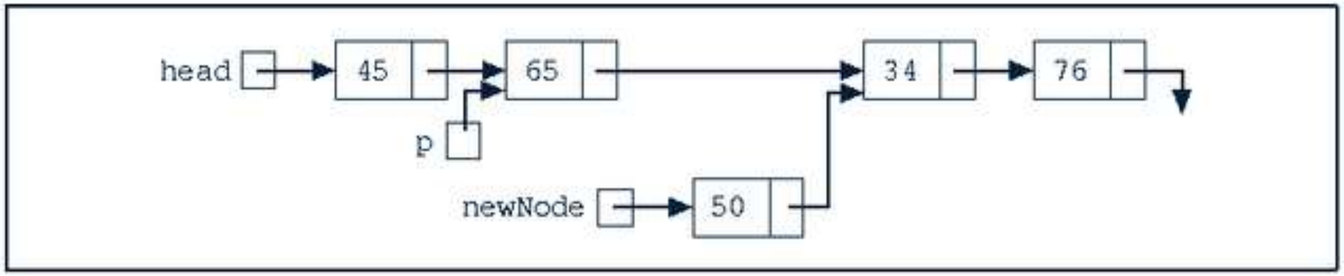


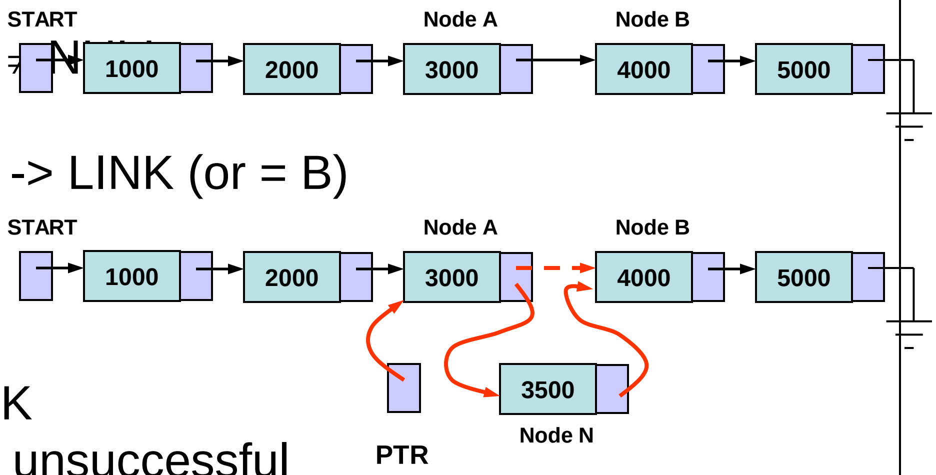
Figure 5-9 Linked list after the statement `newNode->link = p->link;` executes

*** The sequence of events does NOT matter for proper insertion

Insert an Item

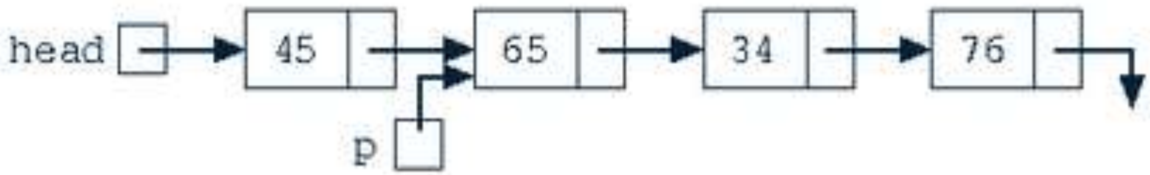
- Insertion into a Listed List
  - Let START be a pointer to a linked list in memory with successive nodes A and B. Write an algorithm to insert node N between nodes A and B.
  - Algorithm

```
1.  Set PTR = START
2.  Repeat step 3 while PTR ≠ NULL
3.  If PTR == A, then
4.          Set N->LINK = PTR -> LINK (or = B)
5.          Set PTR->LINK = N
6.          exit
7.  else
8.          Set PTR=PTR->LINK
9.  If PTR == NULL insertion unsuccessful
10. Stop
```



3 cases: first node, last node, in-between node. (ex: if ITEM = 500? if ITEM = 6000?)

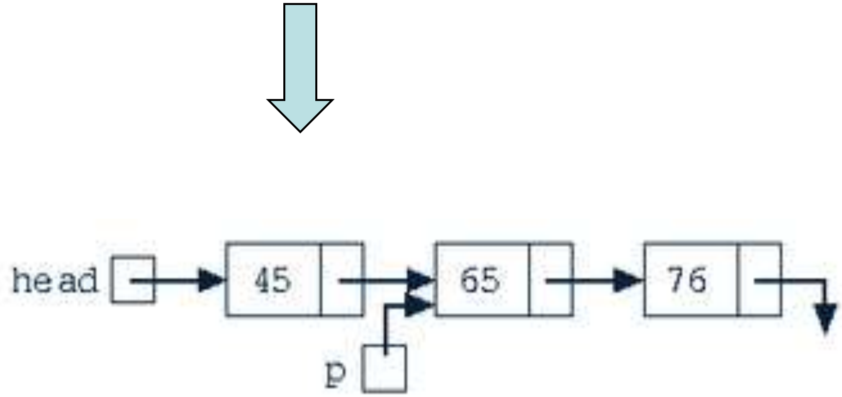
Deletion



Node to be deleted is 34

Deletion

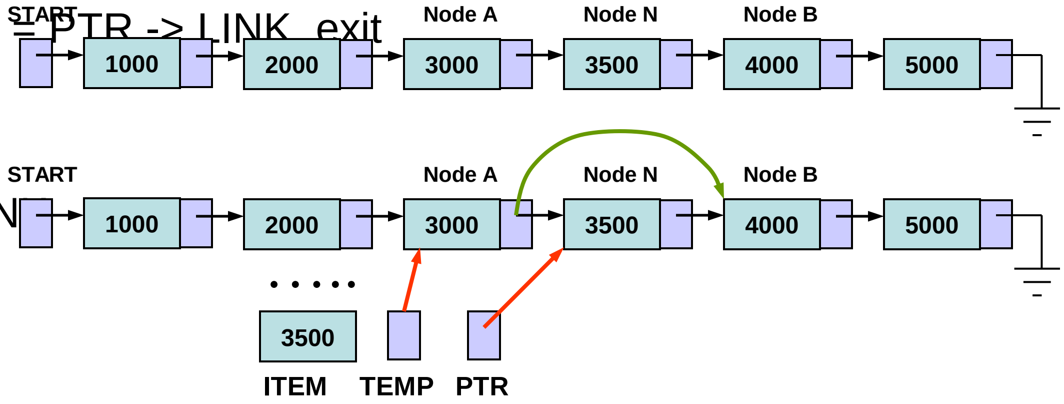
```
q = p->link;
p->link = q->link;
delete q;
```



Delete an Item

- Deletion from a Linked List
  - Let START be a pointer to a linked list in memory that contains integer data. Write an algorithm to delete note which contains ITEM.
  - Algorithm

```
1.  Set PTR=START and TEMP = START
2.  Repeat step 3 while PTR ≠ NULL
3.  If PTR->DATA == ITEM, then
4.          Set TEMP->LINK = PTR -> LINK  exit
5.  else
6.          TEMP = PTR
7.          PTR = PTR -> LIN
8.  Stop
```



Lecture 5

Building a Linked List

- There are two ways to build a linked list

1) forwards

2) backwards

Building a Linked List

What is needed to build a linked list forward:

- -a pointer for the first node
- -a pointer for the last node
- -a pointer for the new node being added

Building a Linked List

- Steps to build a linked list forward:
  - Create a new node called newNode
  - If first is NULL, the list is empty so you can make first and last point to newNode
  - If first is not NULL make last point to newNode and make last = newNode

Building a Linked List

- What is needed to build a linked list backwards
  - a pointer for the first node
  - a pointer to the new node being added

Building a Linked List

- Steps to build a linked list backwards:
  - Create a new node newNode
  - Insert newNode before first
  - Update the value of the pointer first

Linked List ADT

- Basic operations on a linked list are:
    - Initialize the list
    - Check whether the list is empty
    - Output the list
    - Find length of list
    - Destroy the list

Linked List ADT

- Basic operations on a linked list are:
  - Get info from last node
  - Search for a given item
  - Insert an item
  - Delete an item
  - Make a copy of the linked list

Ordered Link List

- In an ordered linked list the elements are sorted
- Because the list is ordered, we need to modify the algorithms (from how they were implemented for the regular linked list) for the search, insert, and delete operations

Lecture 6

Doubly Linked List

- A doubly linked list is a linked list in which every node has a next pointer and a back pointer
- Every node (except the last node) contains the address of the next node, and every node (except the first node) contains the address of the previous node.
- A doubly linked list can be traversed in either direction

Doubly Linked List

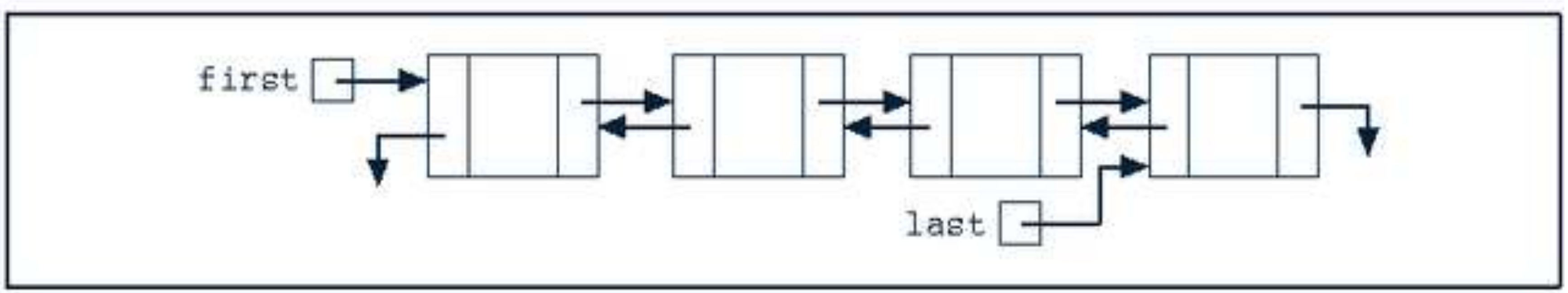


Figure 5-42 Doubly linked list

STL Sequence Container: List

- List containers are implemented as doubly linked lists

Linked Lists With Header and Trailer Nodes

- One way to simplify insertion and deletion is never to insert an item before the first or after the last item and never to delete the first node
- You can set a header node at the beginning of the list containing a value smaller than the smallest value in the data set
- You can set a trailer node at the end of the list containing a value larger than the largest value in the data set

Linked Lists With Header and Trailer Nodes

- These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list.
- The actual list is between these two nodes.

Lecture 7

Circular Linked List

- A linked list in which the last node points to the first node is called a circular linked list

- In a circular linked list with more than one node, it is convenient to make the pointer first point to the last node of the list

Circular Linked List

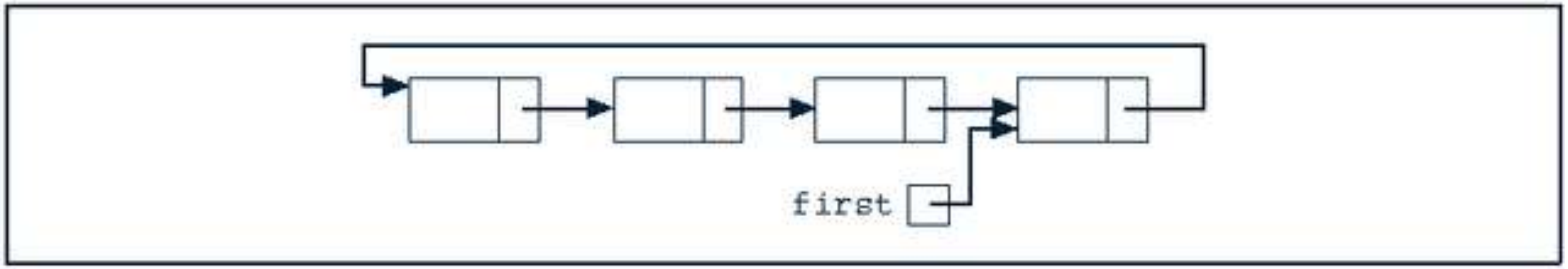
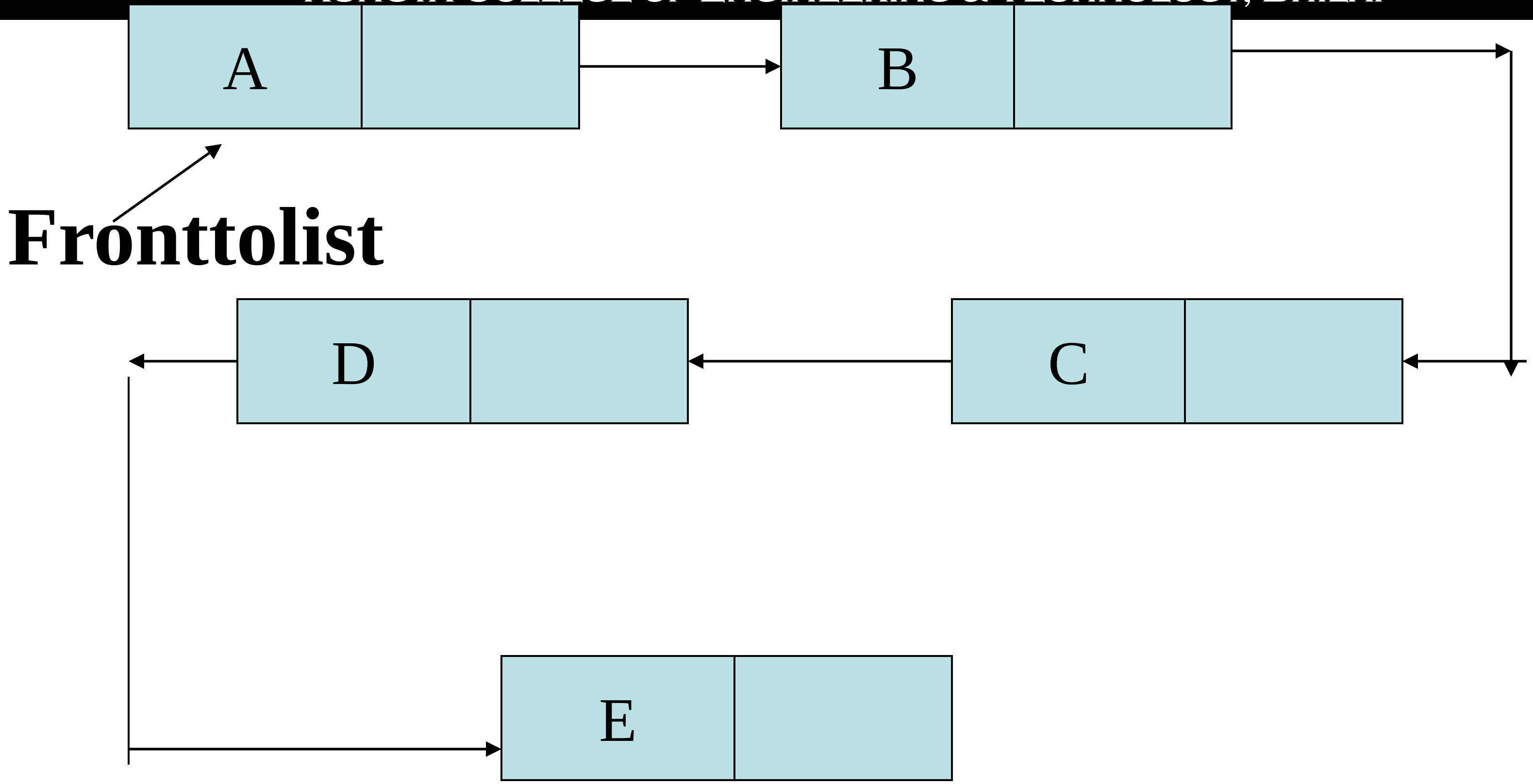
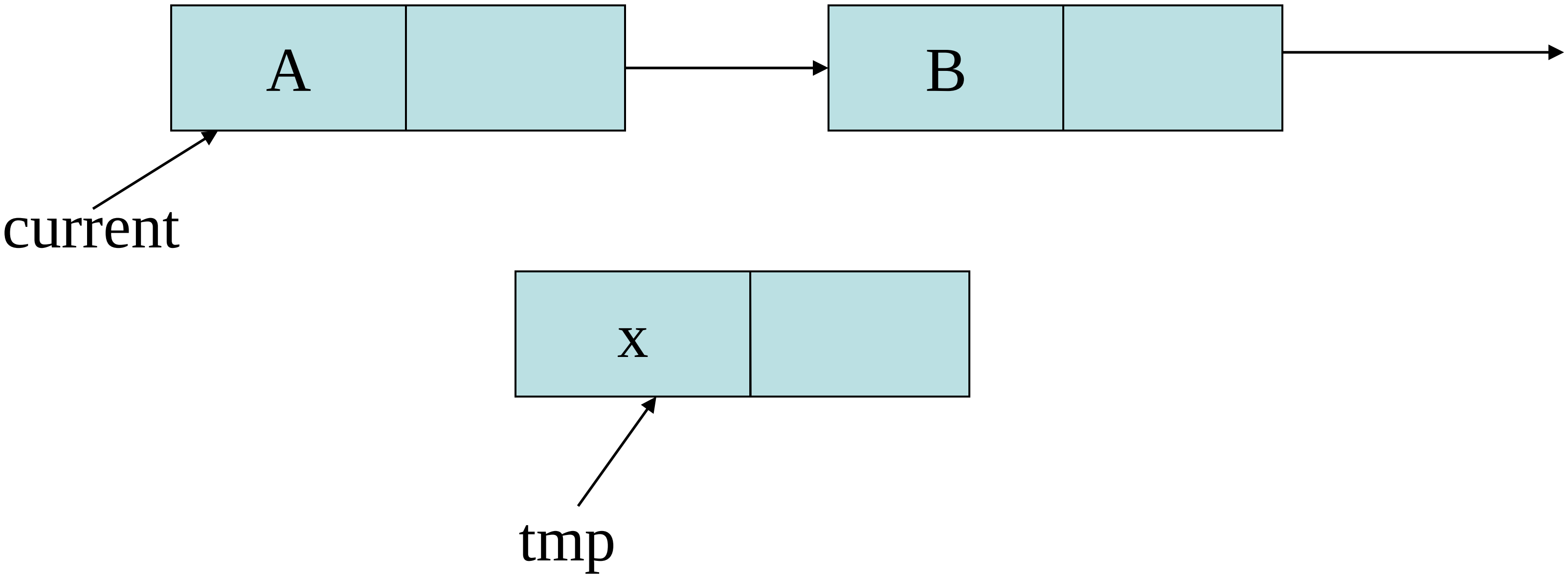


Figure 5-52 Circular linked list with more than one node



The first node in the linked list is accessible through by a reference,we can print or search in the linked list by starting at the first item and following the chain of the references. Insertion and deletion can be performed arbitrary.

linked list



We must perform the following steps:

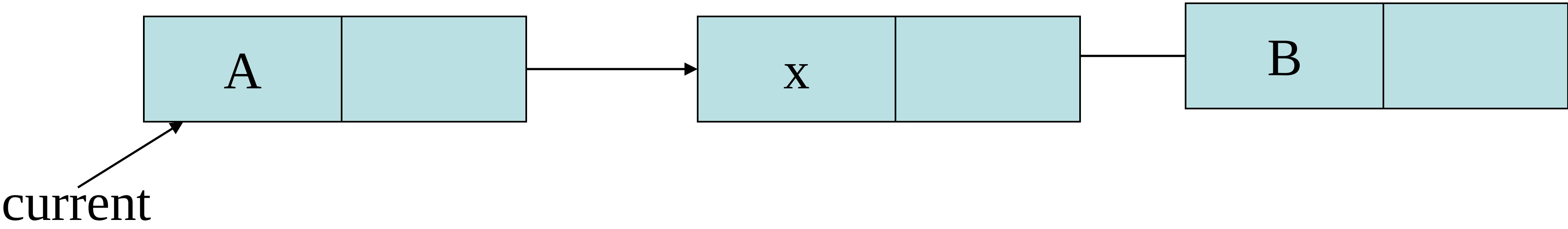
Tmp=new ListNode( );// create a new node

Tmp.element=x; //place x in the element field.

Tmp.next=current.next; // x's next node is B

Current.next=temp; //a's next node is x

Deletion from a linked list



The remove command can be executed in one reference change. To remove element x from the linked list;we set current to be the node prior to x and then have current's next reference by pass x.

current.next=current.next.next

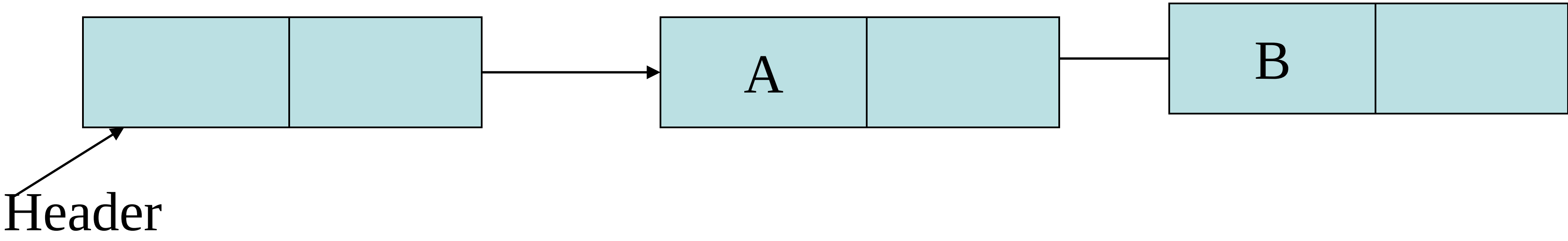
The list A,X,B now appear as A,B

Header Nodes

- If you want to delete item x, then we set current to be node prior to x and then have current's next reference by pass x.
- If you are trying to delete 1st element it becomes a special case
- Special cases are always problematic in algorithm design and frequently leads to bugs in the code.
- It is generally preferable to write code that avoids special cases.

- One way to do that here is to introduce the header node
- A header node is an extra node in the linked list that holds no data serves to satisfies the requirement that every node that contains an item have a previous node in the list

Example



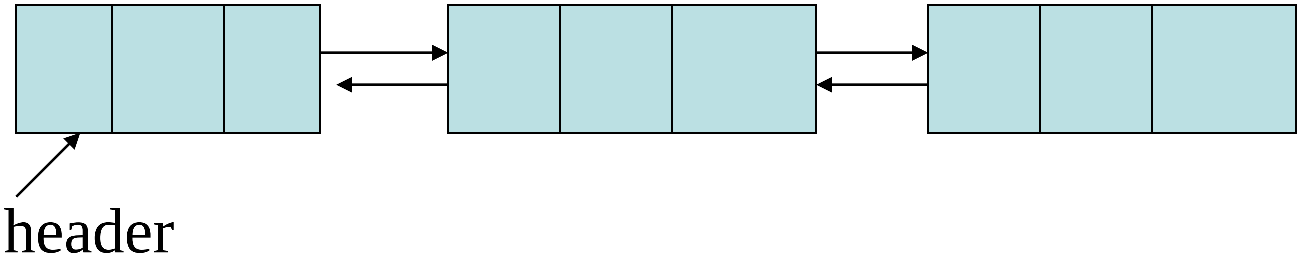
Moving to the front now means setting the current position to header node,and so-on, with a header node, a list is empty then header.next is null.

Ex: implementation of primitive operations with a header node.

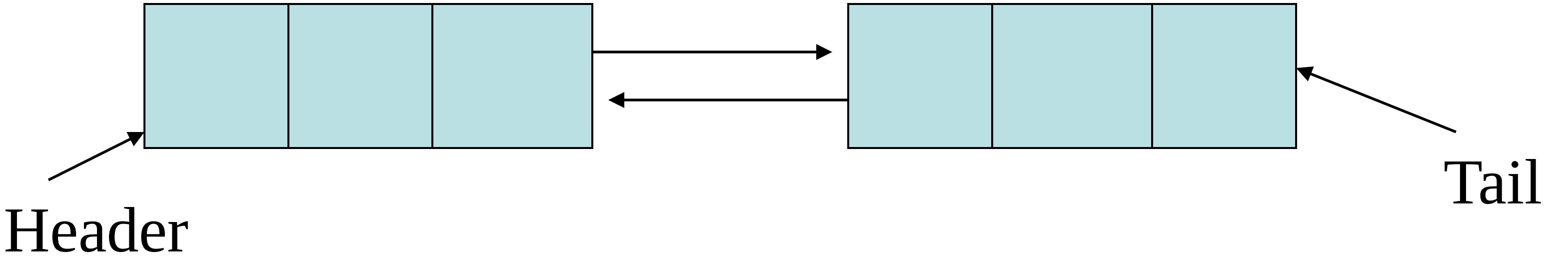
Lecture 8

Doubly linked list and circular linked lists.

- There are two references one for the forward and other one for the backward direction.We should have not only a header but also a tail.



If list is empty, then list consists of head and tail connected together.



Empty doubly linked list

If doubly linked list is empty then

Header.next==tail;

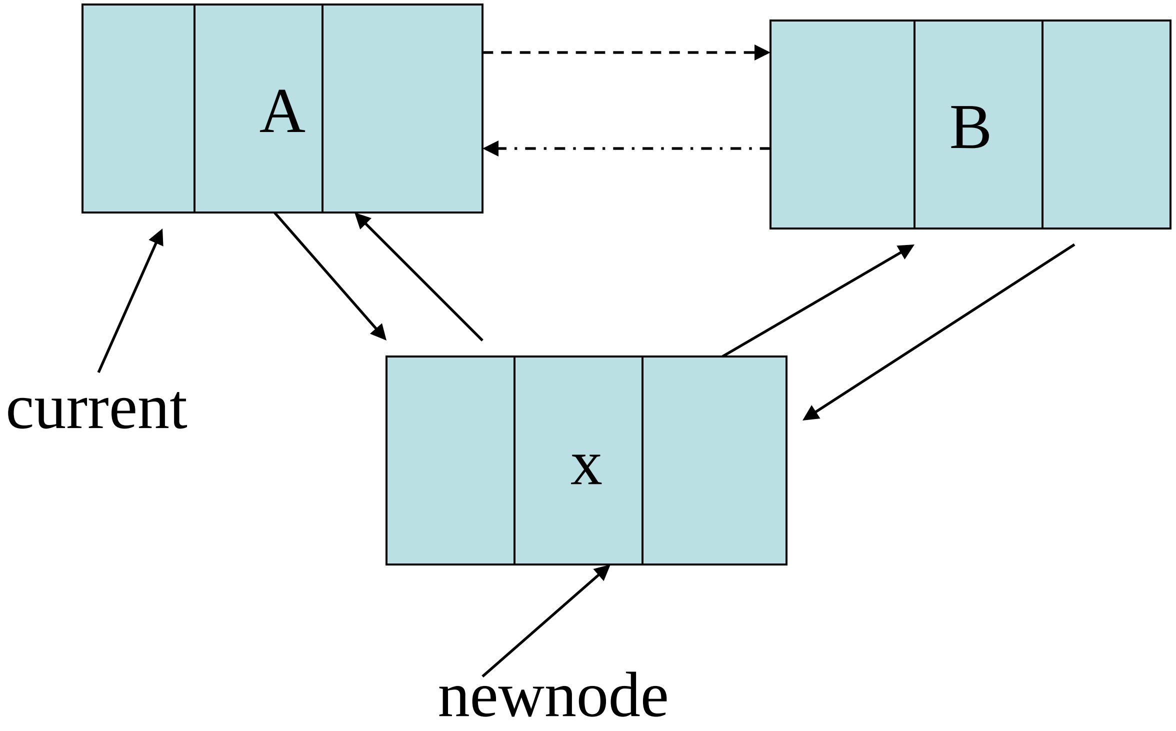
Or

Tail.prev==Header

The doubly linked list class is shown below

```
Class DoublyLinkedListNode
{
Object data //some element
DoublyLinkedListNode next
DoublyLinkedListNode  prev
}
```

Insert operation

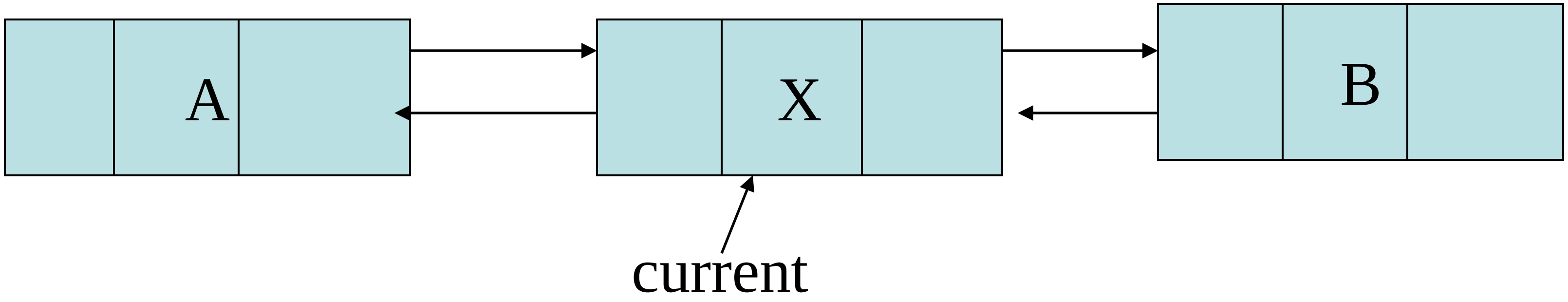


Algorithm

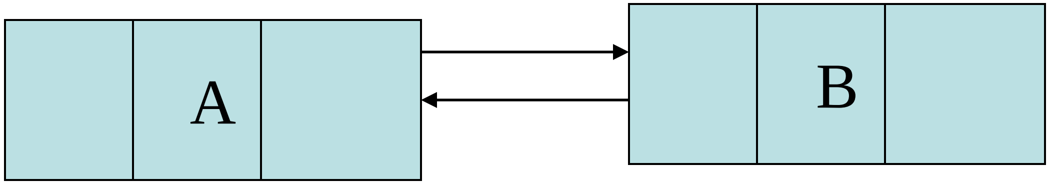
```
1. Newnode=new DoublyLinkedListNode(x);
2. Newnode.prev=current;
3. Newnode.next=current.next;
4. Newnode.prev.next=newnode;
5. Newnode.next.prev=newnode;
```

Delete operation

Before deletion



After deletion



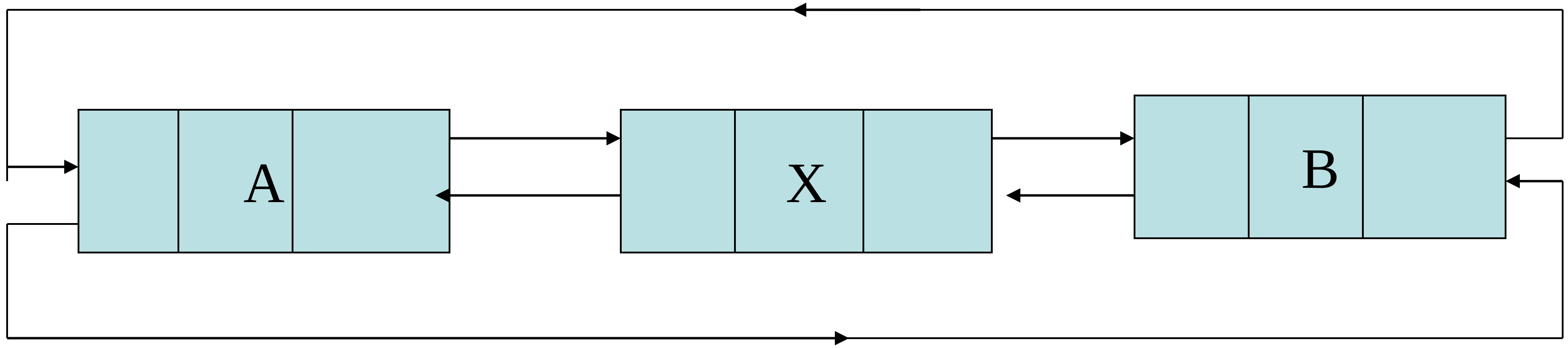
Delete operation

1. Current.prev.next=current.next
2. Current.next.prev=current.prev
3. Current=head

Circular doubly linked list.

- A popular convention is to create a circular doubly linked list , in which the last cell keeps a reference back to the first and first cell keeps a back reference to the last cell.
- This can be done with or without a header.

Circular doubly linked list

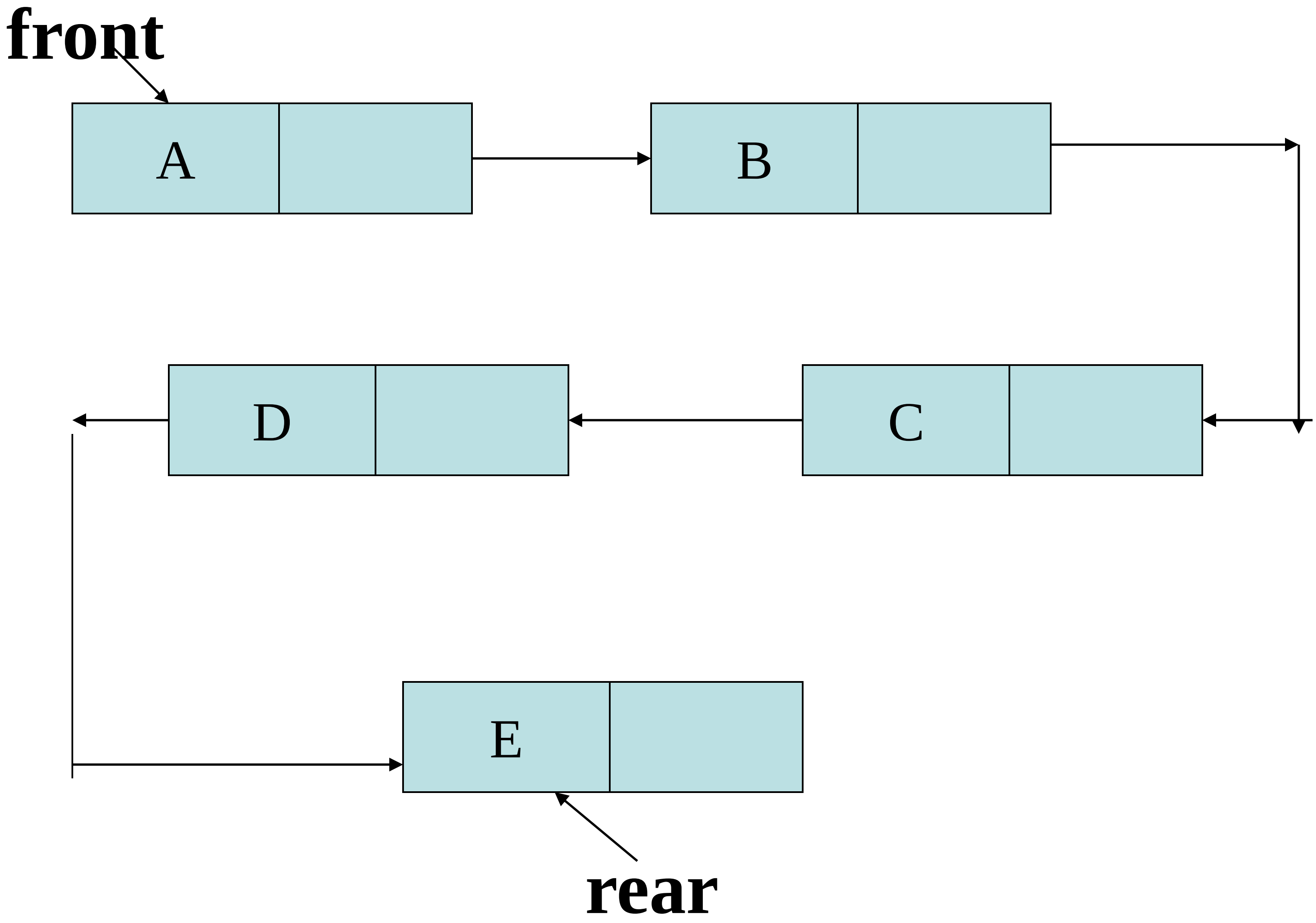


Ex: implementation of circular doubly liked list operations

Linked list representation of queues.

- The queue can be implemented by a linked list, provided references are kept to both the **front** and **rear** of the list.

Example



The queue is almost identical to the stack routines.

Public Queue Operations

- Public operations
- //void enqueue(x)- Insert x
- //Object getfront( )-Return least recently inserted

  item
- //object Dequeue( )-> Return and remove least

  recent item
- //Boolean idEmpty( ) -> Return true if empty, false otherwise
- //void MakeEmpty( )-> Remove all items

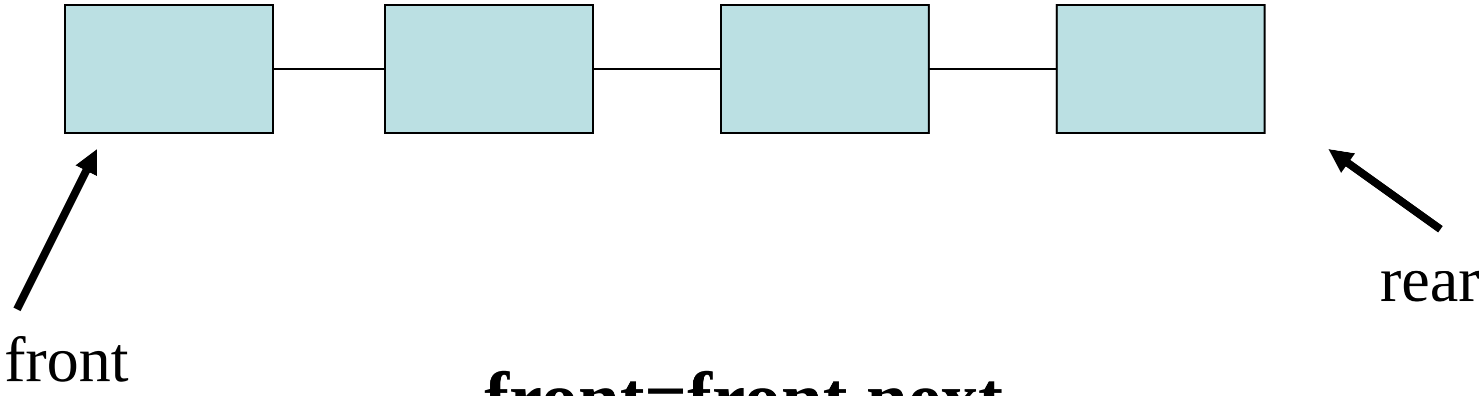
isEmpty( )

```
public Boolean isEmpty( )
{
return front==null;
}
```

makempty()

```
Public void makeEmpty( )
{
front=back=null;
}
```

Dequeue for the linked list based queue class



front=front.next

Dequeue for the linked list based queue class

```
Public Object dequeue( ) throws Underflow
{
//return and remove the least recently inserted item from the queue
//exception Underflow if the queue is empty
If(isempty( )) throw new underflow (“queue dequeue”);
Object returnvalue=front.element;
front=front.next;
return returnvalue;
}
```

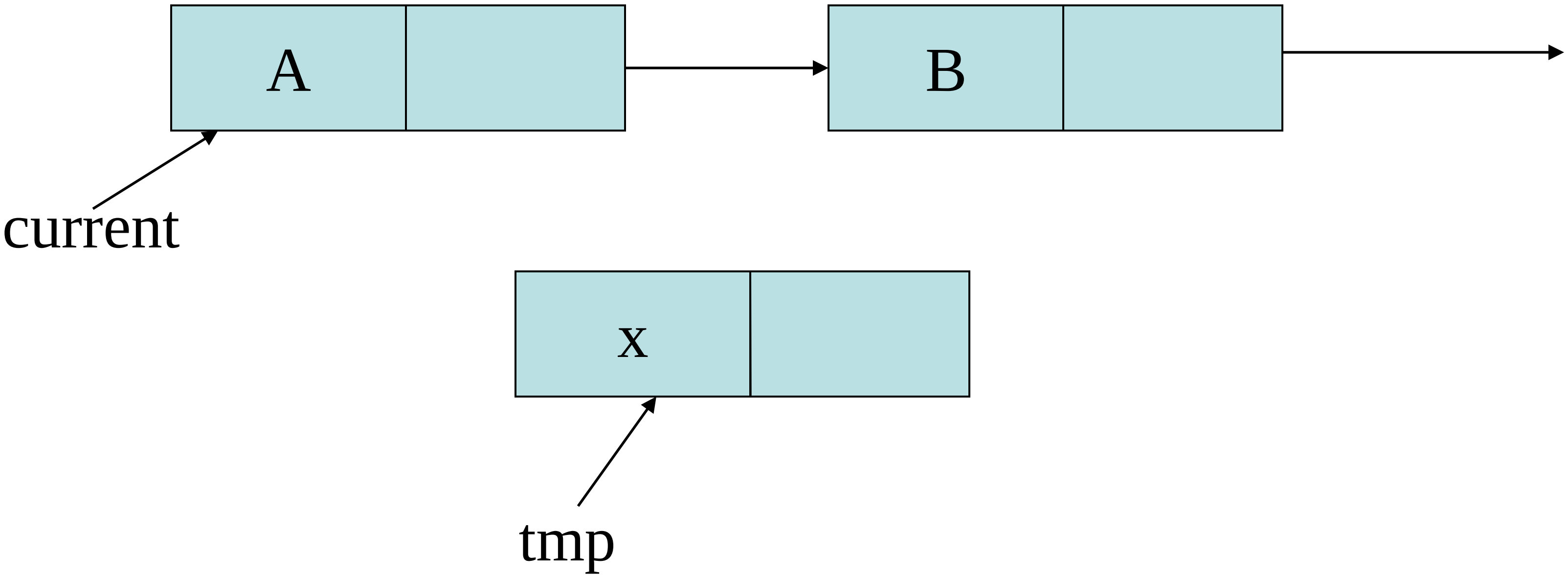
getfront for the linked list based queue class

```
Public Object getfront( ) throws underflow
//get the least recently inserted item in the queue. Does not alter the queue
//exception underflow if queue is empty
If(isempty( ) )
Throw new underflow (“queue getfront”);
Return front.element;
}
```

enqueue for the linked list based queue class

```
Public void enqueue(Object x)
//insert a new item into the queue
{
If(isempty( ))
front=rear=new ListNode(x)
else
rear=rear.next=new ListNode(x);
}
```

Ex:Comparison of array based queue and linked list based queues linked list



We must perform the following steps:

Tmp=new ListNode( );// create a new node

Tmp.element=x; //place x in the element field.

Tmp.next=current.next; // x’s next node is B

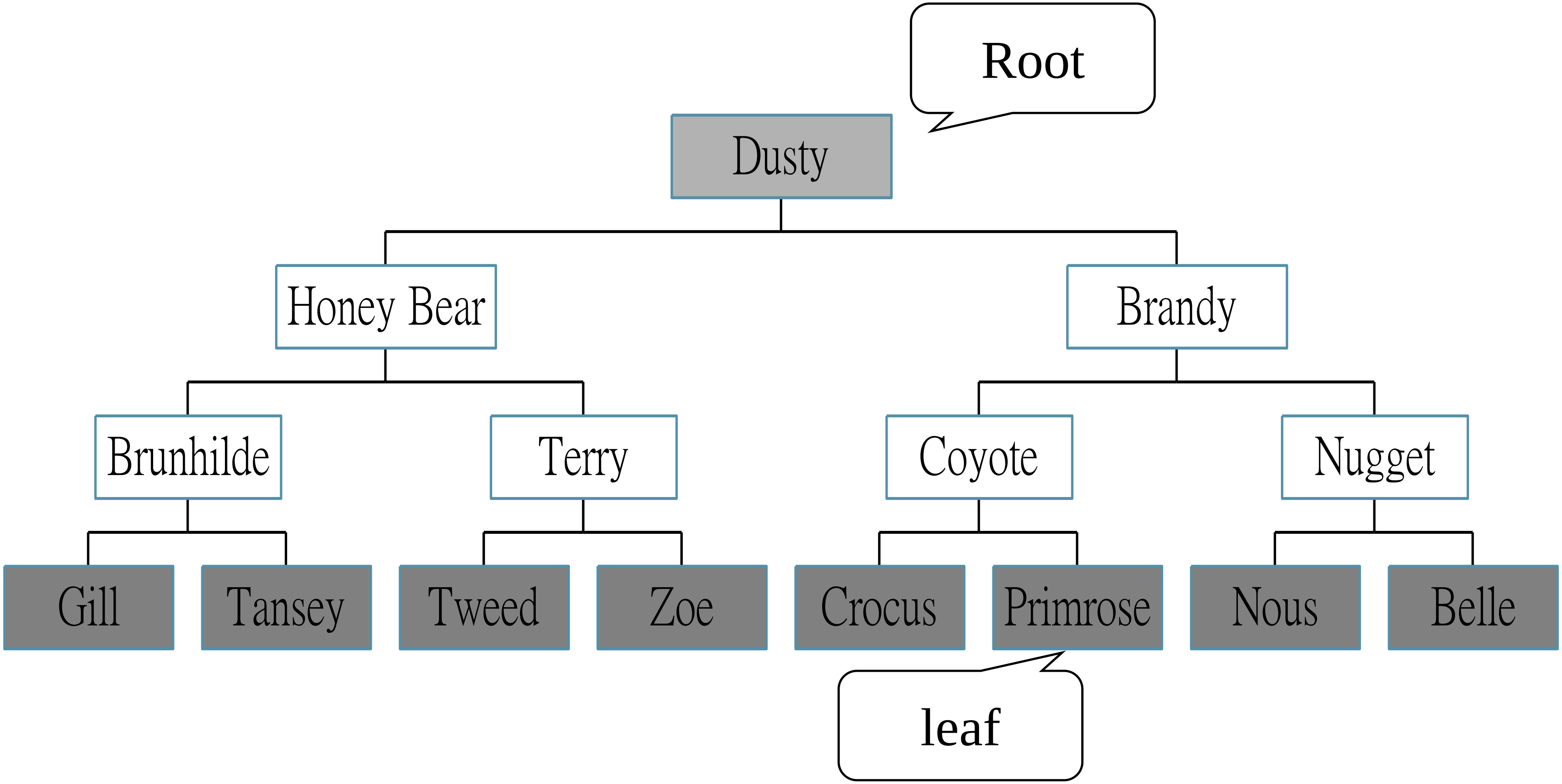
Current.next=temp; //a’s next node is x

UNIT 5

Trees and Graph

Lecture 1

Trees



Root

Dusty

Honey Bear

Brandy

Brunhilde

Terry

Coyote

Nugget

Gill

Tansey

Tweed

Zoe

Crocus

Primrose

Nous

Belle

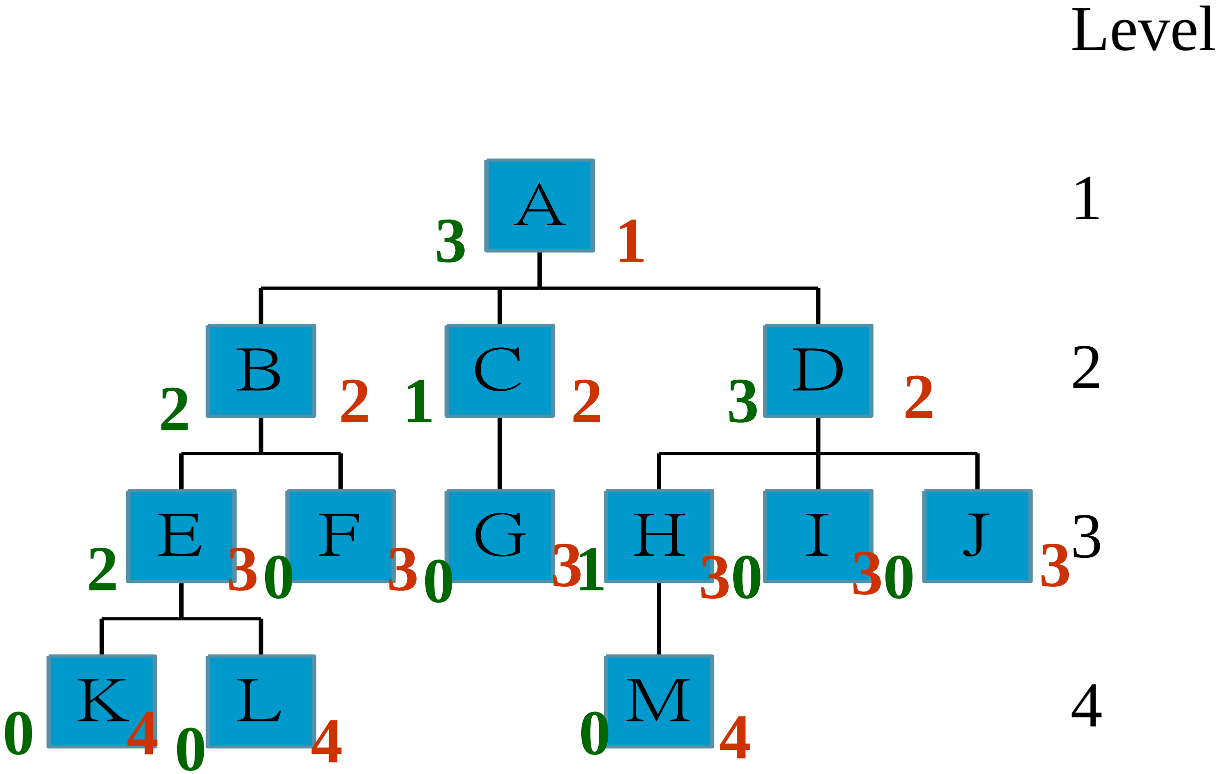
leaf

Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called
  the root.
- The remaining nodes are partitioned into $n>=0$ disjoint sets T_1, ..., T_n, where each of these sets is a tree.
- We call T_1, ..., T_n the subtrees of the root.

Level and Depth

node (13)
degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
level of a node
height of a tree (4)



Terminology

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node

Representation of Trees

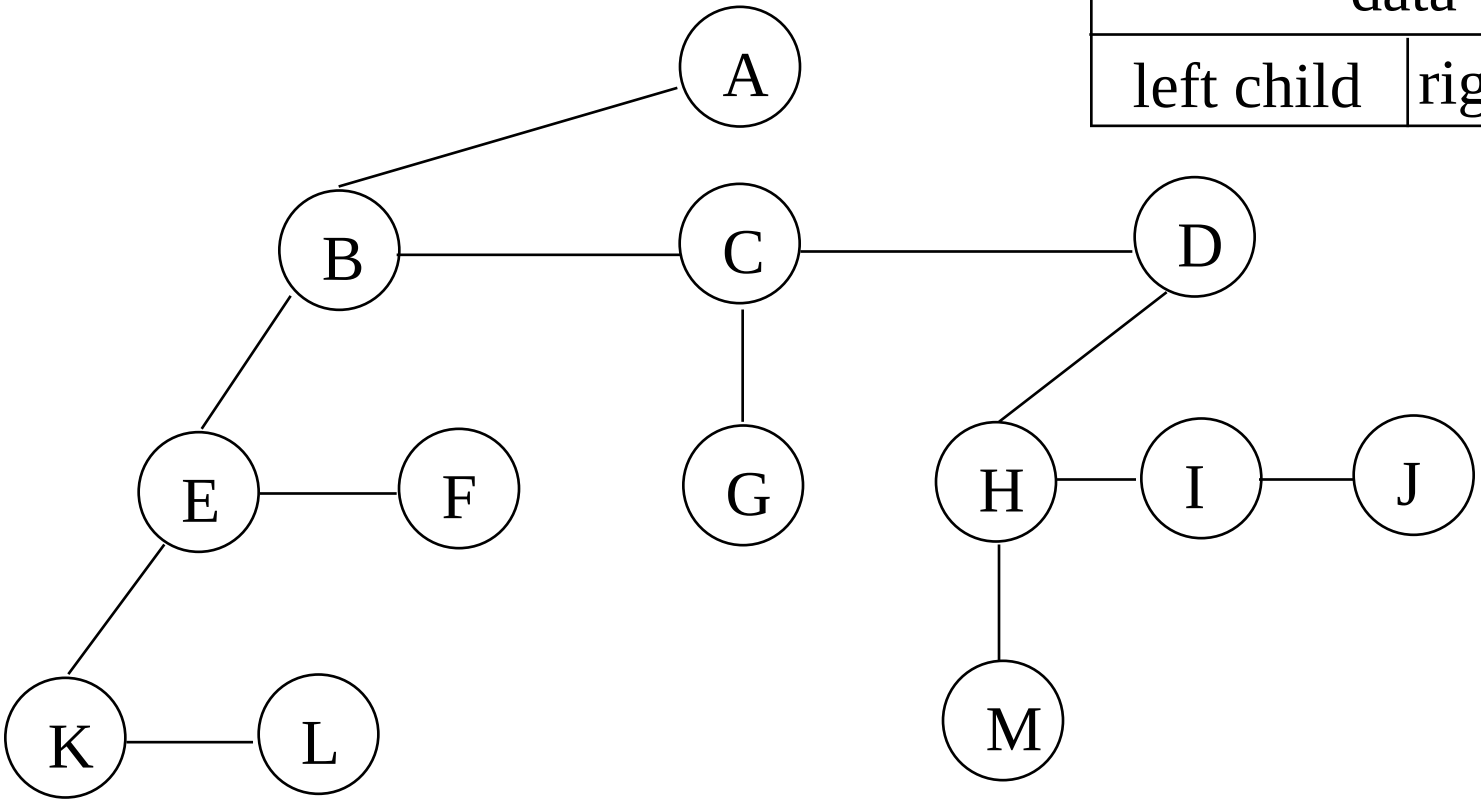
- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)
  - The root comes first, followed by a list of sub-tr

data	link 1	link 2	...	link n

How many link fields are needed in such a representation?

Left Child - Right Sibling

data	
left child	right sibling

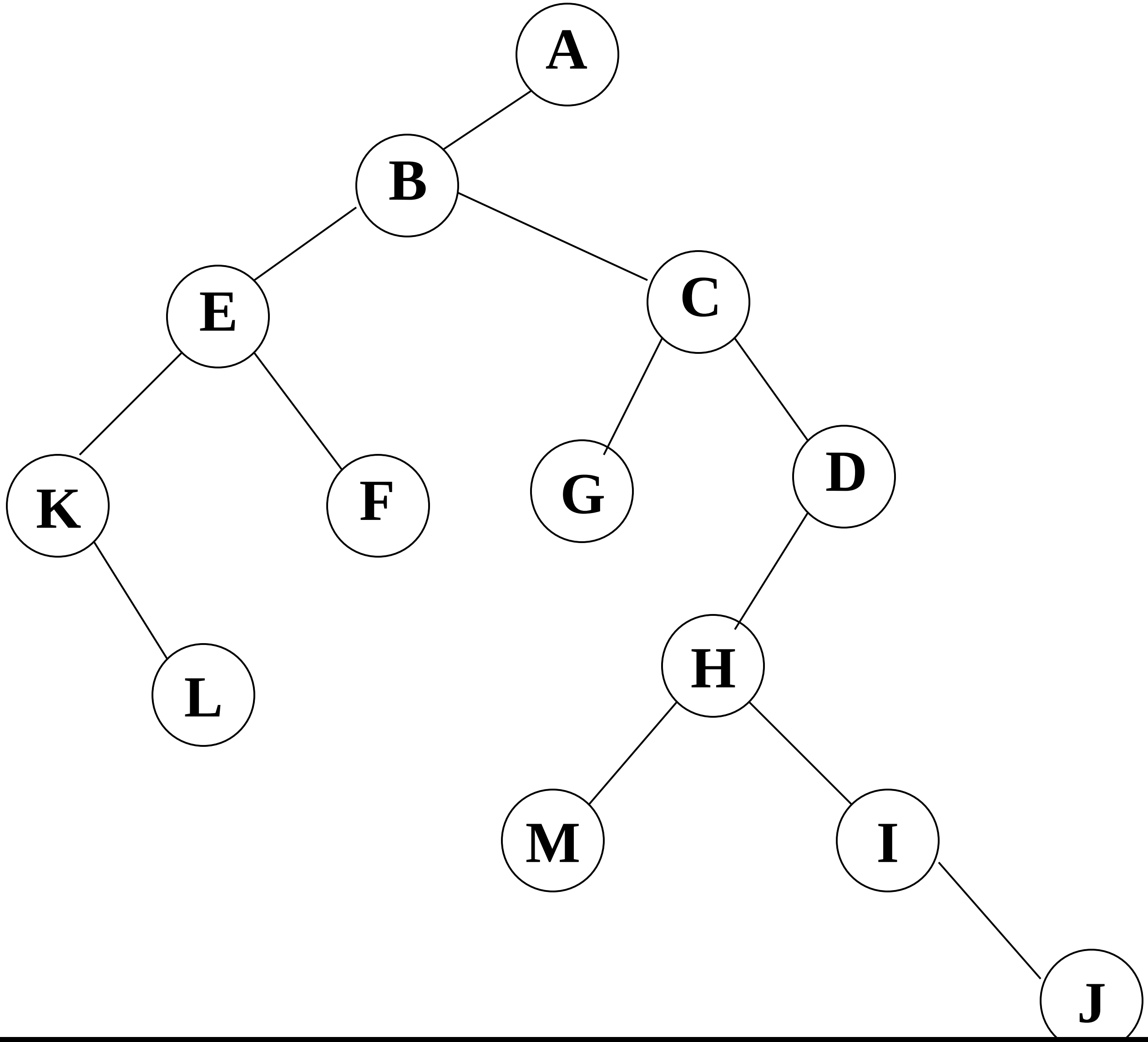


Lecture 2

Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - – by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)



Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary

tree) return *TRUE* else return *FALSE*

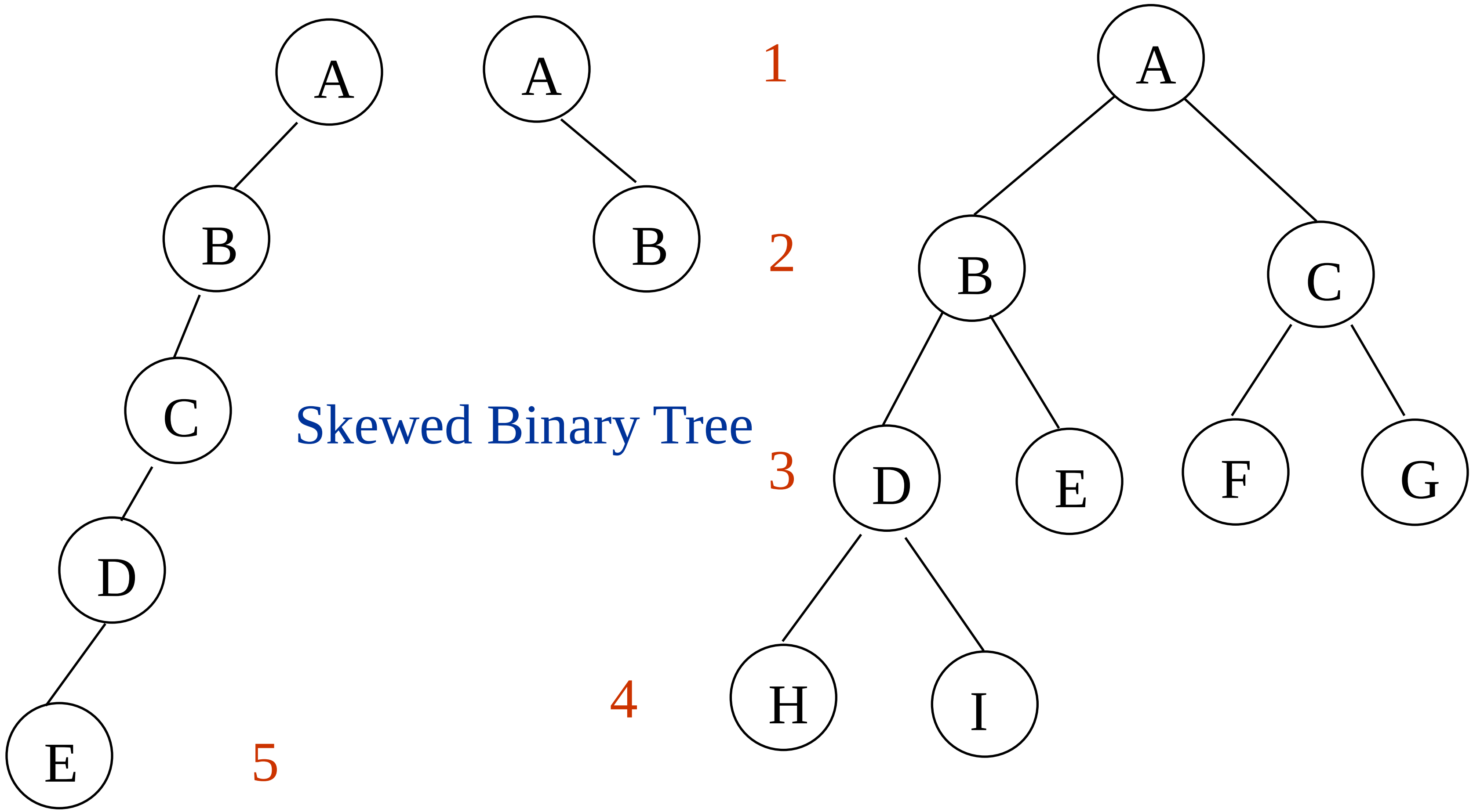
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Samples of Trees

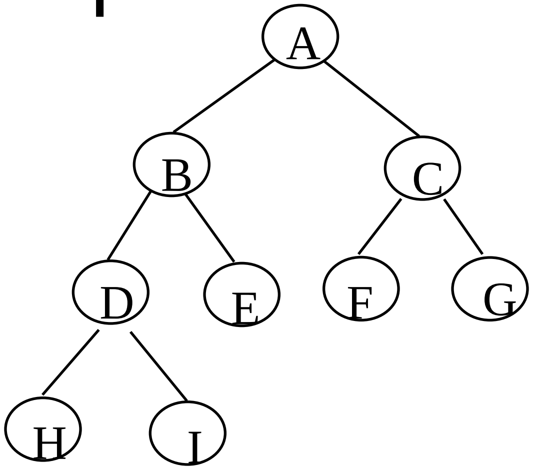


Complete Binary Tree

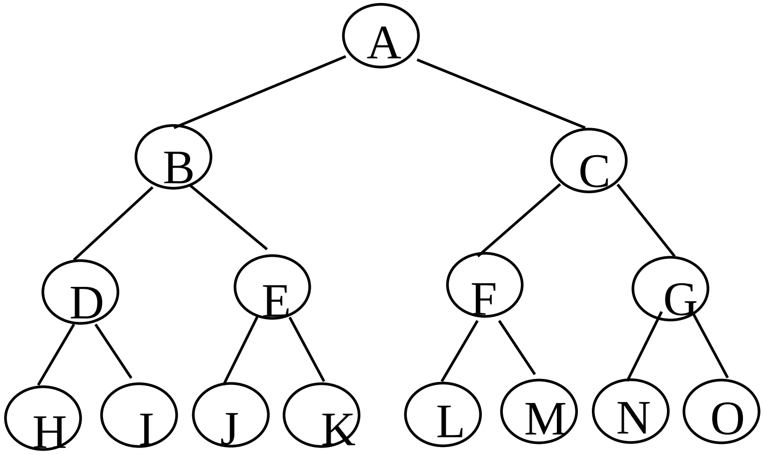
Skewed Binary Tree

Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree o depth *k* having 2^k -1 nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nod numbered from 1 to *n* in the full binary tree o depth *k*.



Complete binary tree

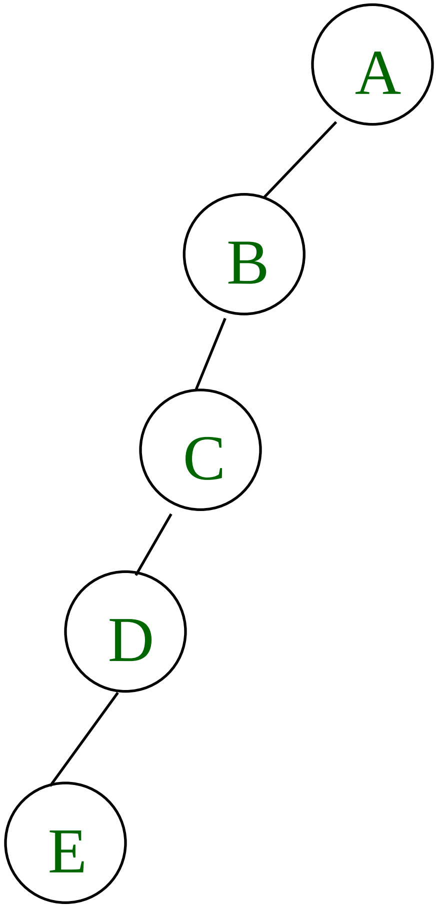


Full binary tree of depth 4

Binary Tree Representations

- If a complete binary tree with n nodes (depth = log n + 1) is represented sequentially, then for any node with index i, $1<=i<=n$, we have:
  - *parent*(i) is at $i/2$ if $i!=1$. If $i=1$, i is at the root and has no parent.
  - *left_child*(i) ia at $2i$ if $2i<=n$. If $2i>$n, then i has

Sequential Representation

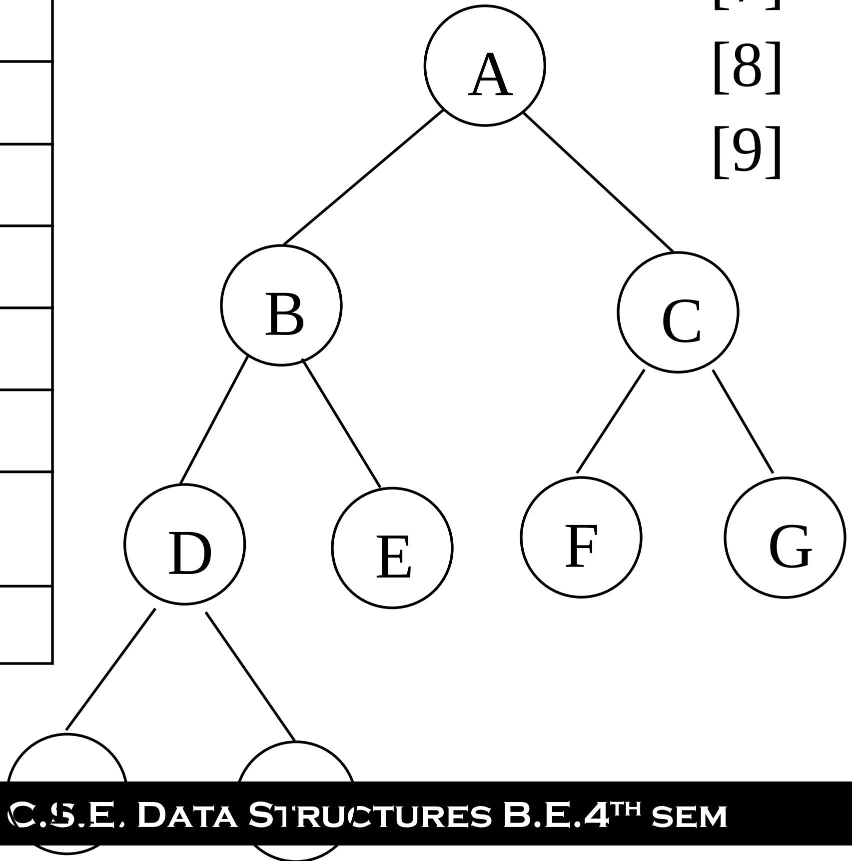


[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space

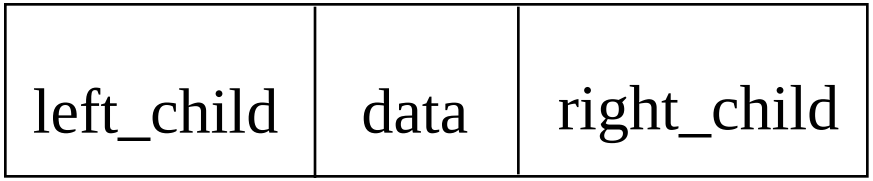
(2) insertion/deletion problem

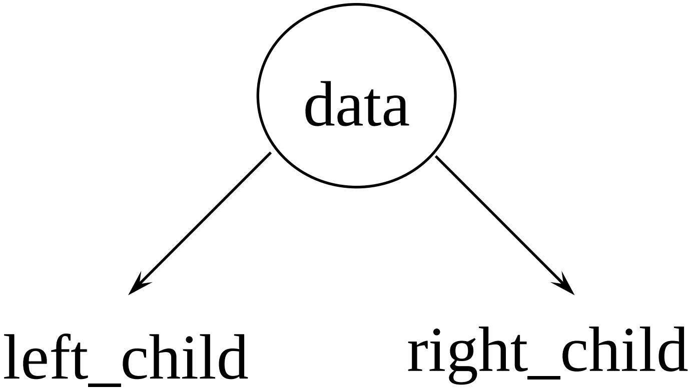
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I



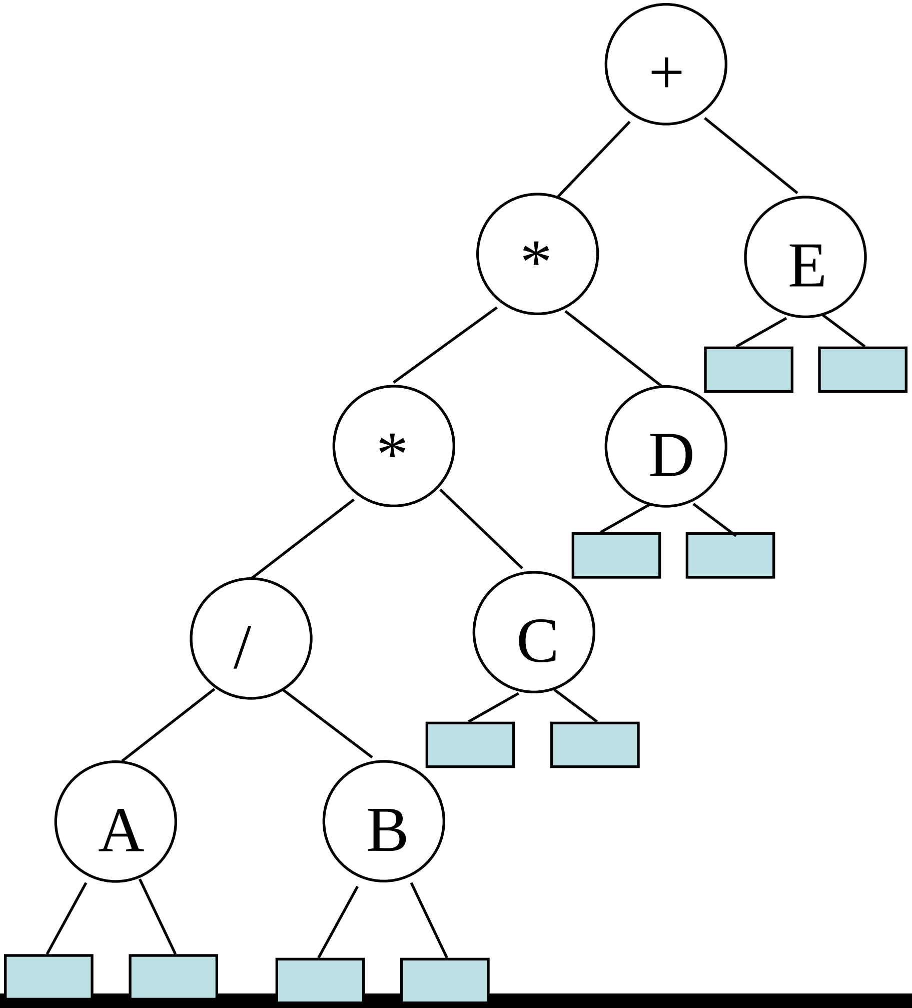
Linked Representation

```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```





Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

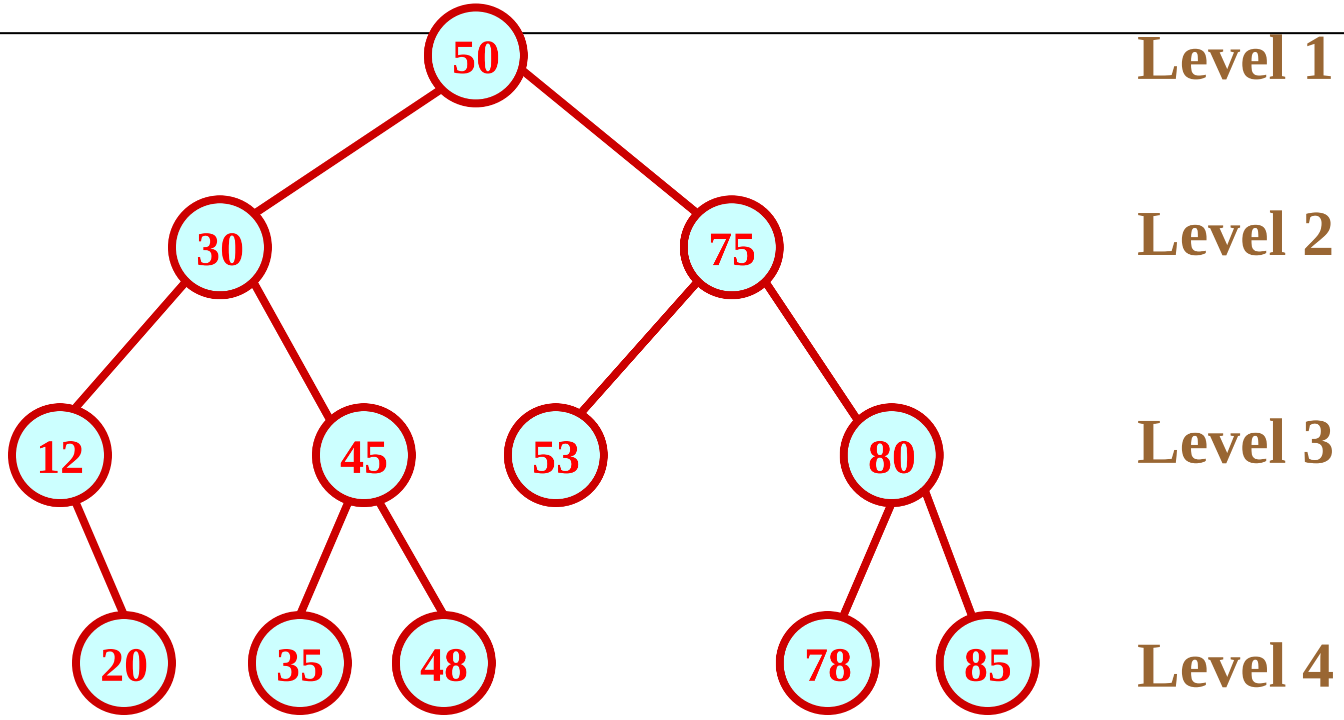
A B / C * D * E +

postfix expression

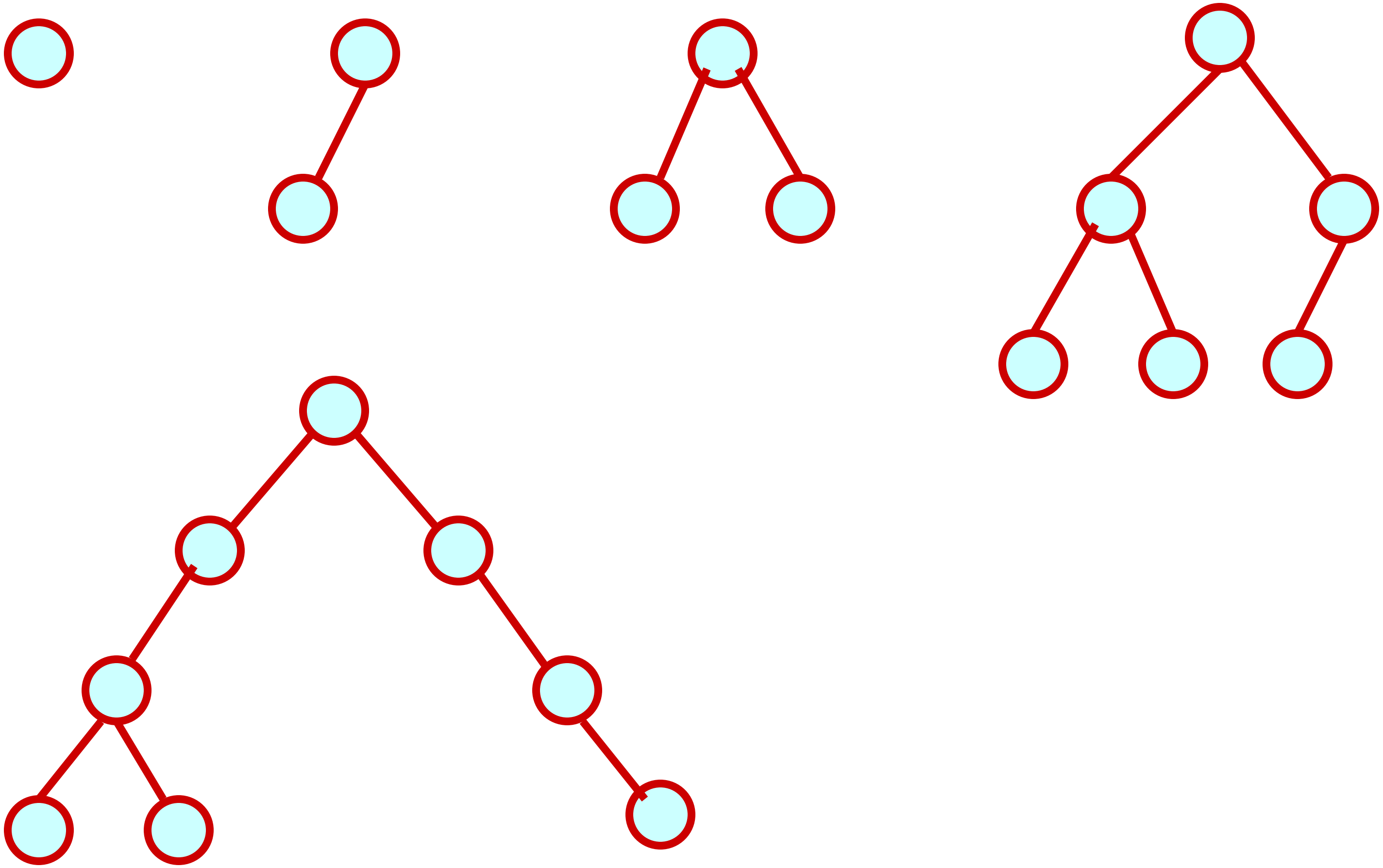
level order traversal

+ * E * D / C A B

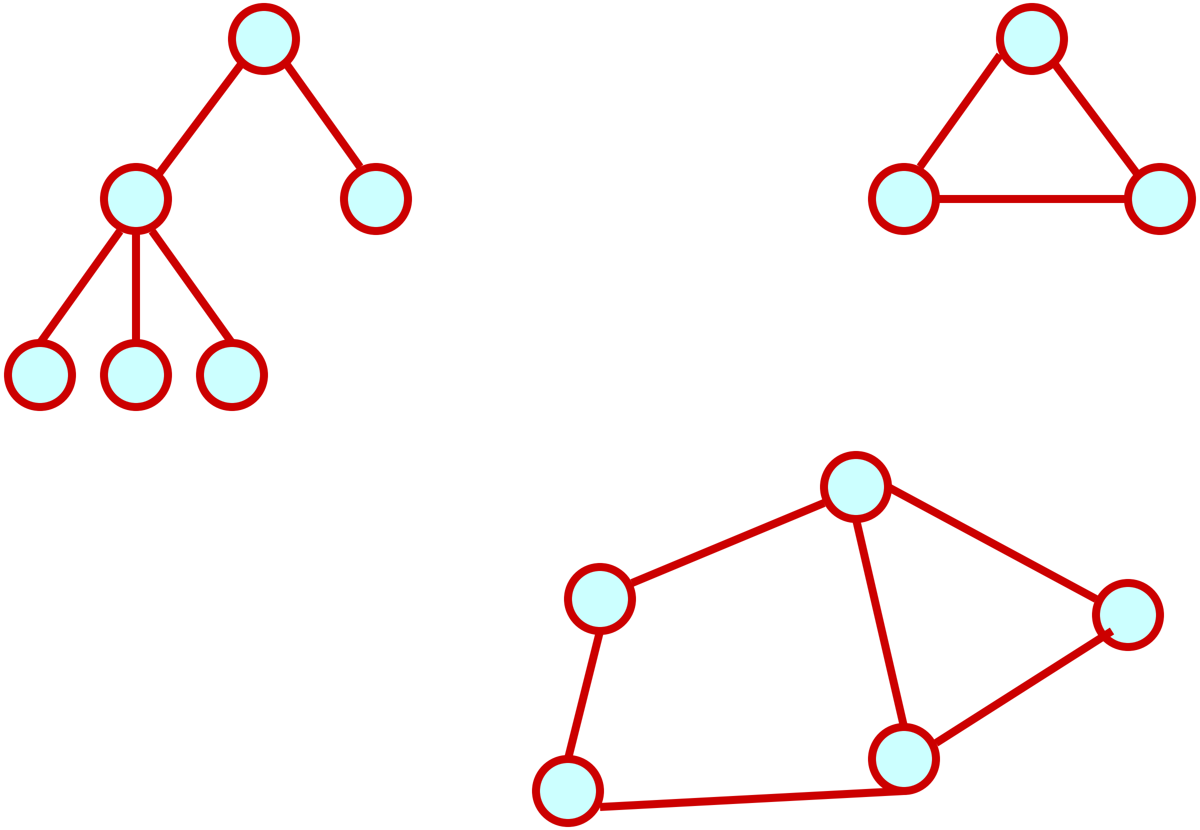
Illustration



Examples of Binary Trees



Not Binary Trees

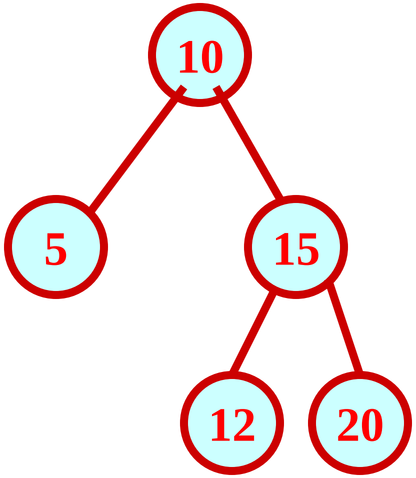


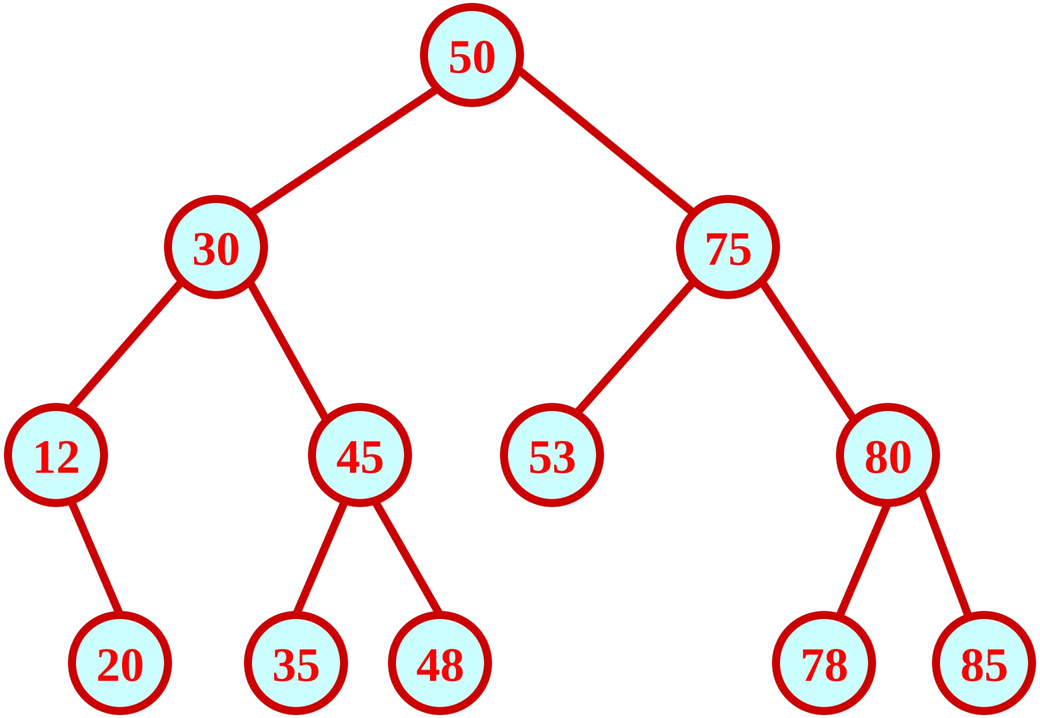
Lecture 3

Binary Search Trees

- A particular form of binary tree suitable for searching.
- Definition
  - A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the following conditions:
    - All keys (if any) in the left subtree of the root precede the key in the root.
    - The key in the root precedes all keys (if any) in its right subtree.
    - The left and right subtrees of the root are again binary search trees.

Examples

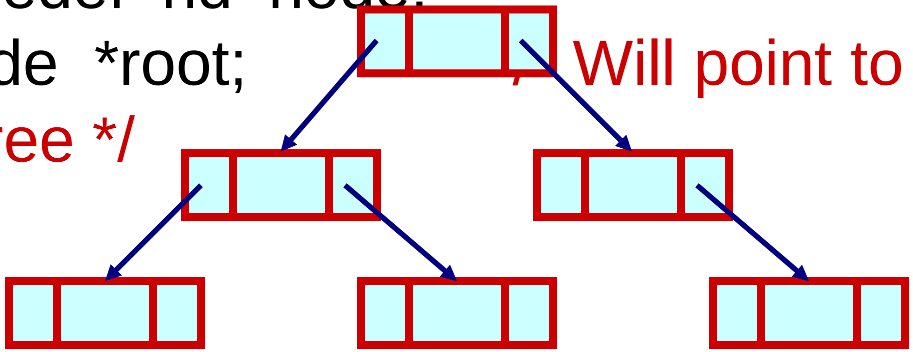




How to Implement a Binary Tree?

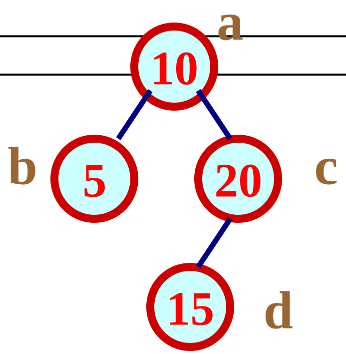
- Two pointers in every node (left and right).

```
struct  nd  {
                int  element;
                struct  nd  *lptr;
                struct  nd  *rptr;
            };
typedef  nd  node;
node  *root;          / Will point to the root of the
  tree */
```



An Example

- Create the tree



```
a  =  (node *) malloc (sizeof (node));
b  =  (node *) malloc (sizeof (node));
c  =  (node *) malloc (sizeof (node));
d  =  (node *) malloc (sizeof (node));
a->element = 10;   a->lptr = b;         a->rptr = c;
b->element = 5;    b->lptr = NULL;  b->rptr = NULL;
c->element = 20;   c->lptr = d;         c->rptr = NULL;
d->element = 15;   d->lptr = NULL;  d->rptr = NULL;
root = a;
```

Traversal of Binary Trees

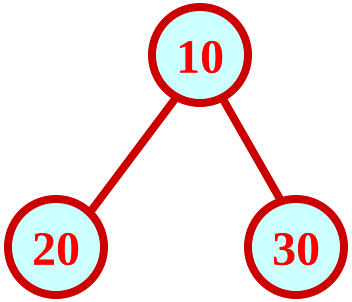
- In many applications, it is required to move through all the nodes of a binary tree, visiting each node in turn.
  - For n nodes, there exists n! different orders.
  - Three traversal orders are most common:
    - Inorder traversal
    - Preorder traversal
    - Postorder traversal

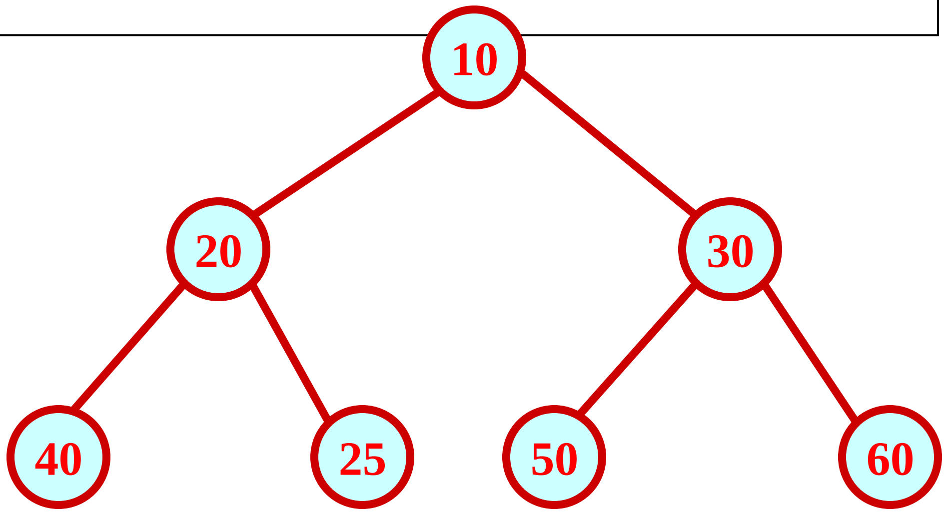
Inorder Traversal

- Recursively, perform the following three steps:
    - Visit the left subtree.
    - Visit the root.
    - Visit the right subtree.

LEFT-ROOT-RIGHT

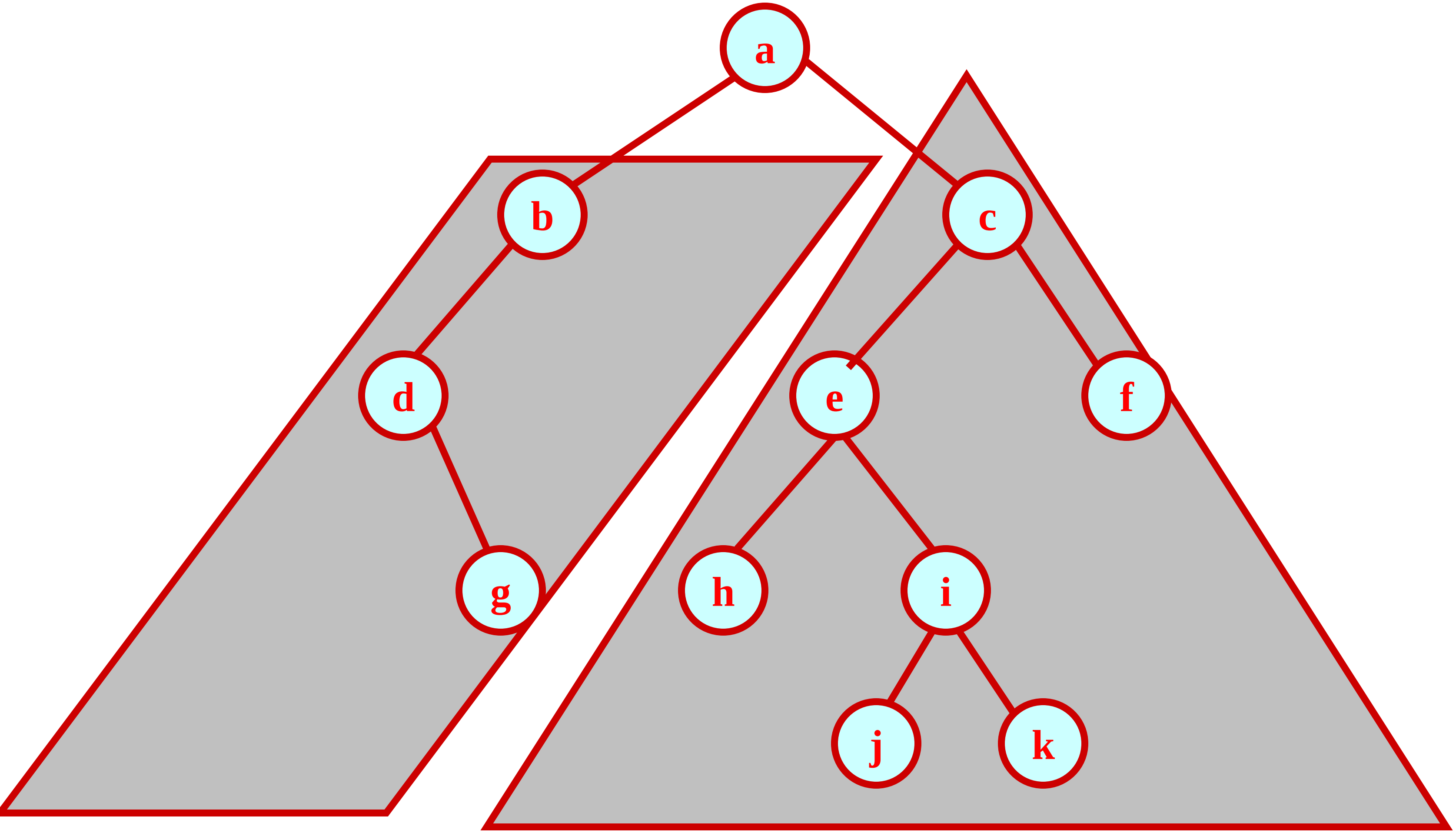
Example:: inorder traversal





20 10 30

40 20 25 10 50 30 60



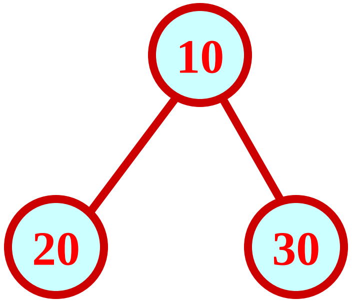
. d g b . a h e j i k c f

Preorder Traversal

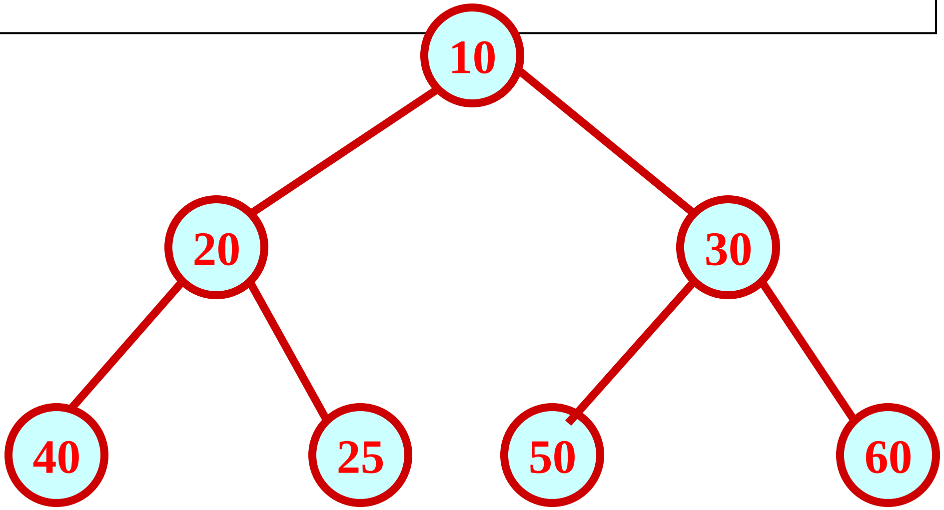
- Recursively, perform the following three steps:
    - Visit the root.
    - Visit the left subtree.
    - Visit the right subtree.

ROOT-LEFT-RIGHT

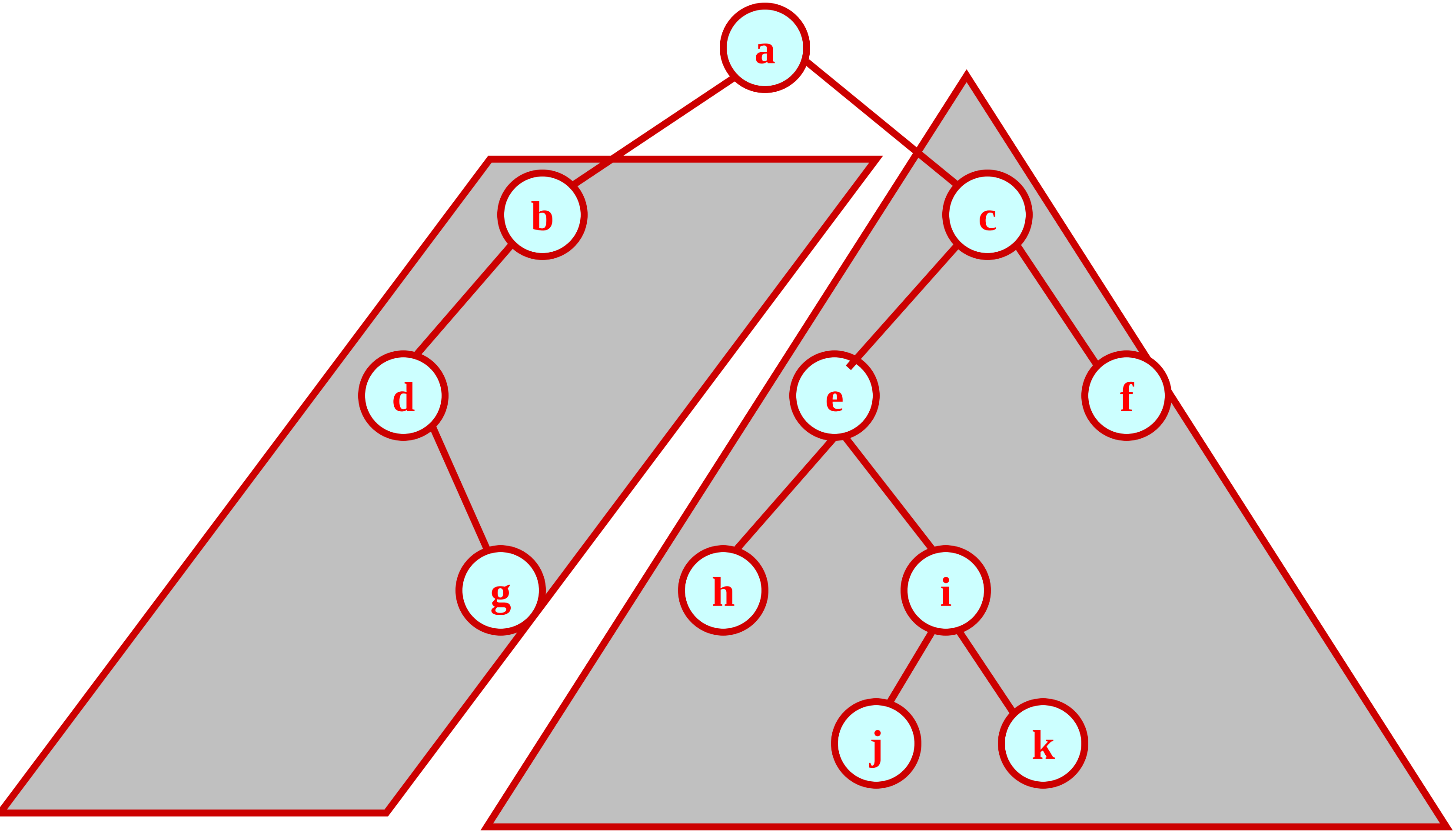
Example:: preorder traversal



10 20 30



10 20 40 25 30 50 60



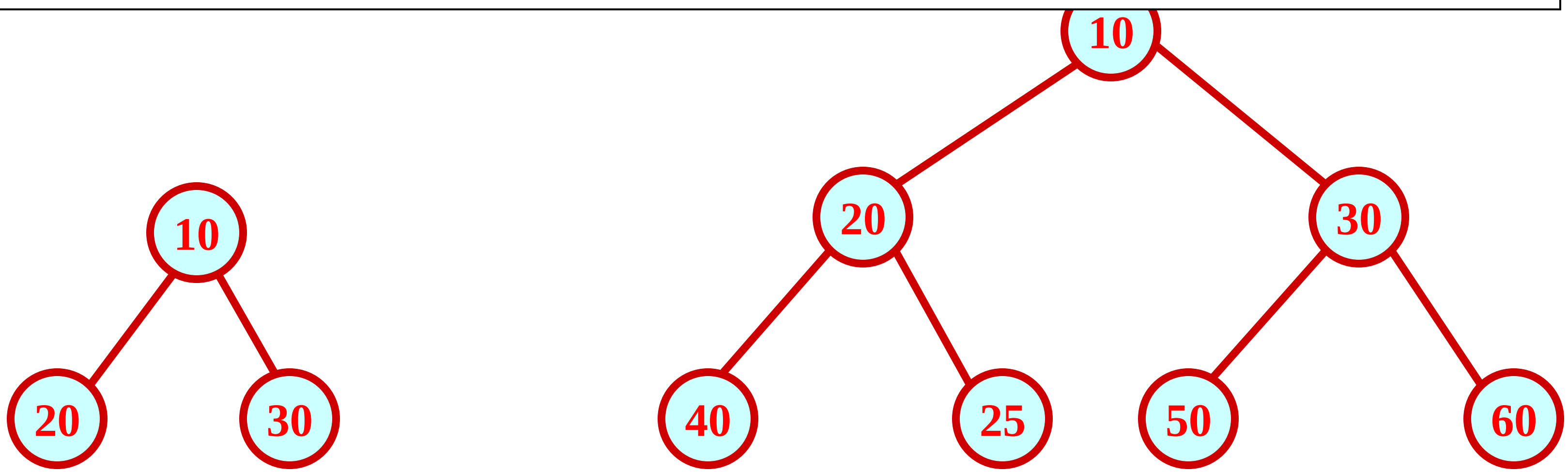
a b d . g . c e h i j k f

Postorder Traversal

- Recursively, perform the following three steps:
  - Visit the left subtree.
  - Visit the right subtree.
  - Visit the root.

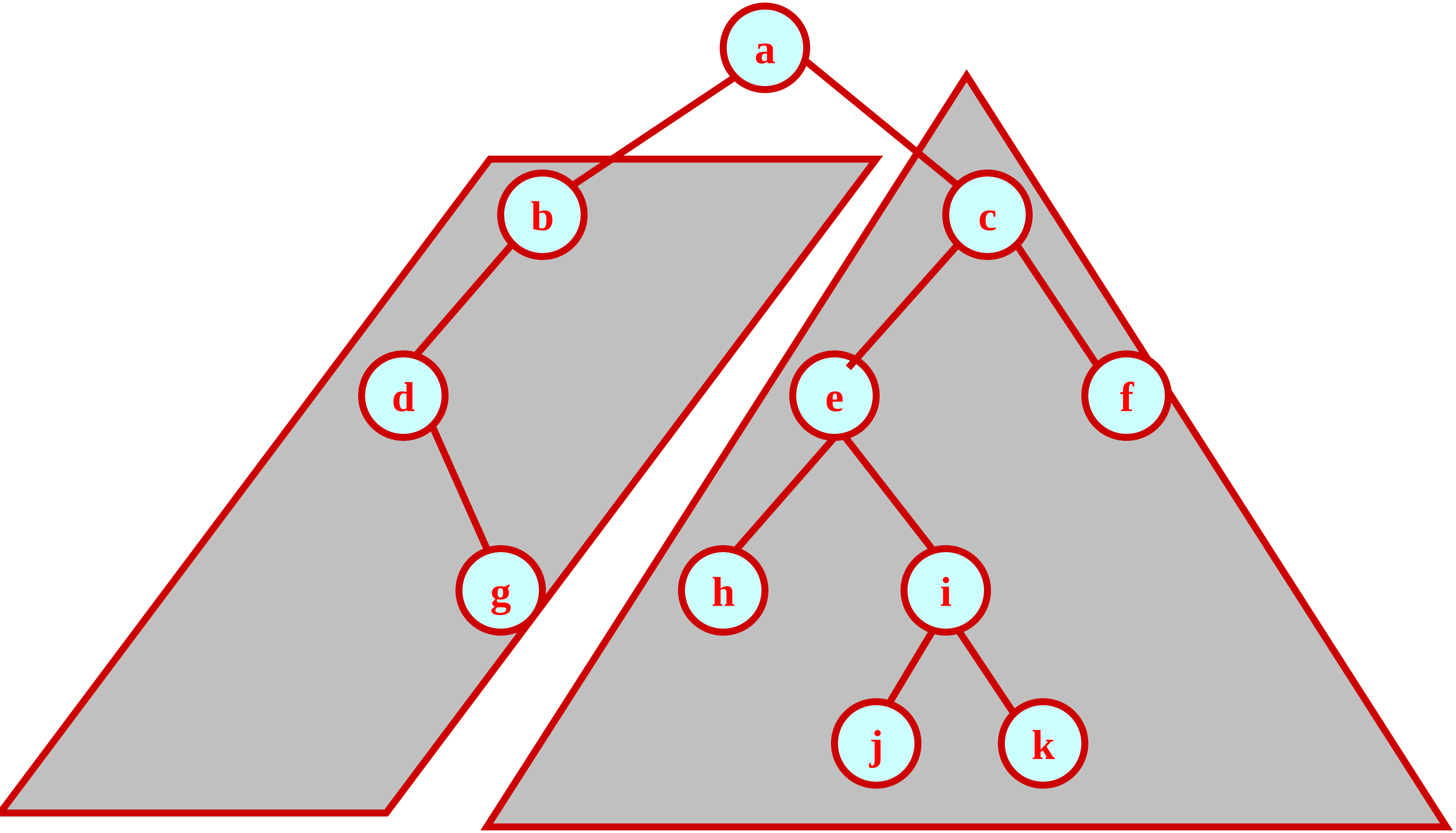
LEFT-RIGHT-ROOT

Example:: postorder traversal



20 30 10

40 25 20 50 60 30 10



. g d . b h j k i e f c a

Implementations

```
 void  inorder  (node  *root)
{
    if  (root != NULL)
    {
         inorder (root->left);
         printf (“%d “, root->element);
         inorder (root->right);
     }
}
```

```
 void  preorder  (node  *root)
{
    if  (root != NULL)
    {
         printf (“%d “, root->element);
         inorder (root->left);
         inorder (root->right);
     }
}
```

```
 void  postorder  (node  *root)
{
    if  (root != NULL)
    {
         inorder (root->left);
         inorder (root->right);
         printf (“%d “, root->element);
     }
}
```

Lecture 4

Threaded Binary Trees

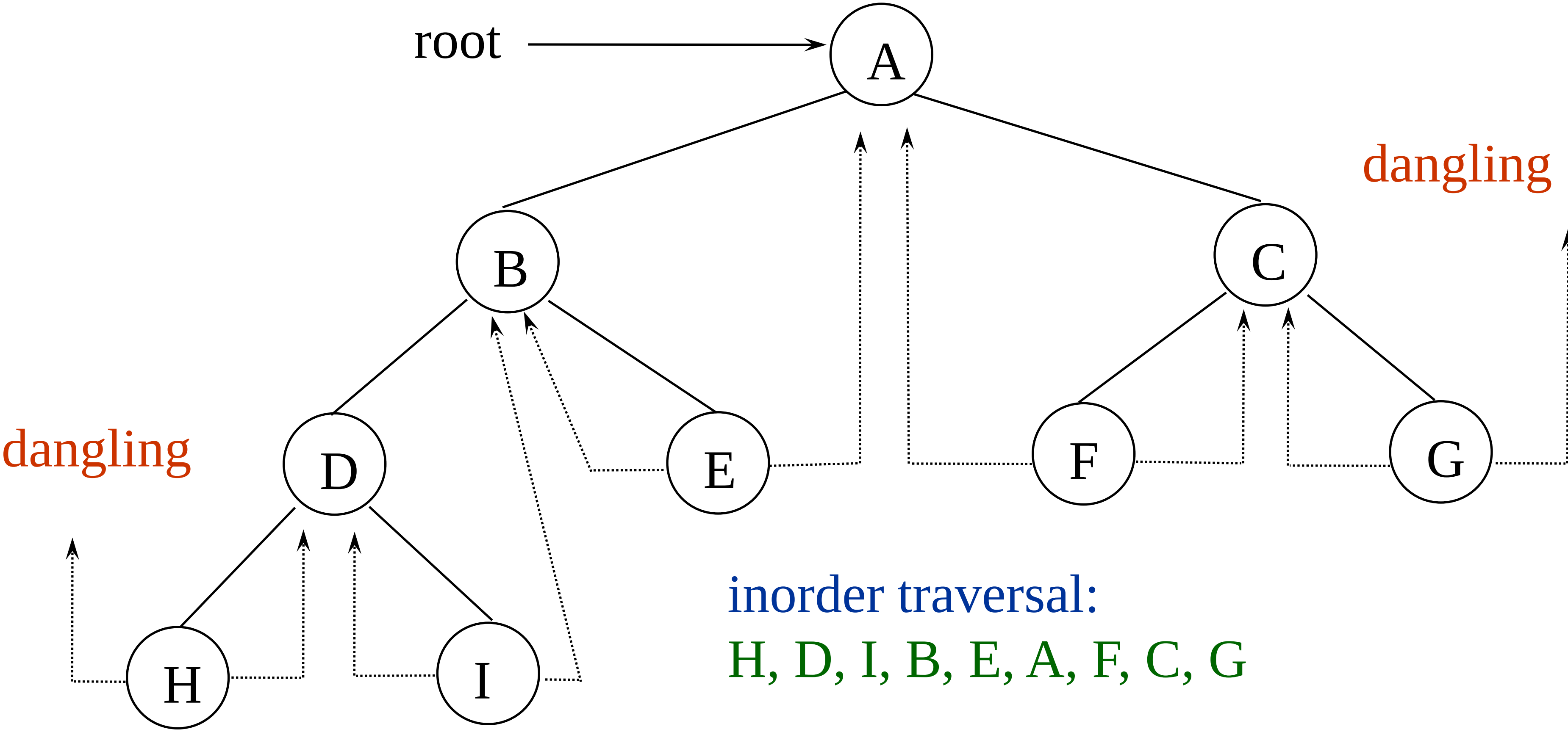
- Two many null pointers in current representation
  of binary trees
  - n: number of nodes
  - number of non-null links: n-1
  - total links: 2n
  - null links: 2n-(n-1)=n+1
- Replace these null pointers with some usefu
  “threads”.

Threaded Binary Trees (*Continued*)

If `ptr->left_child` is null,
replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*

If `ptr->right_child` is null,
replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*

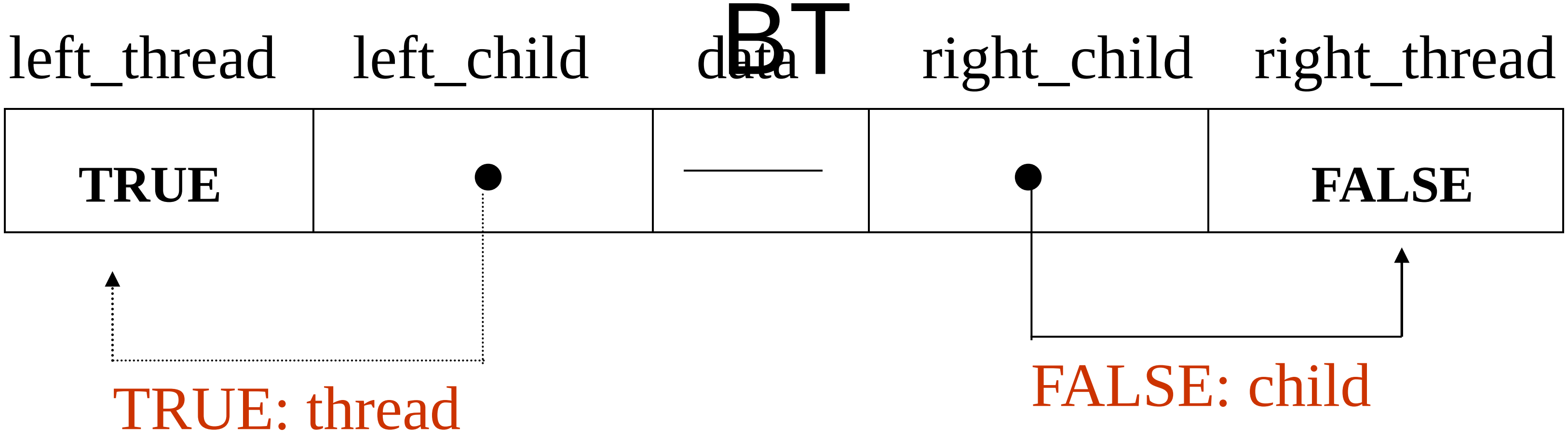
A Threaded Binary Tree



inorder traversal:

H, D, I, B, E, A, F, C, G

Data Structures for Threaded BT

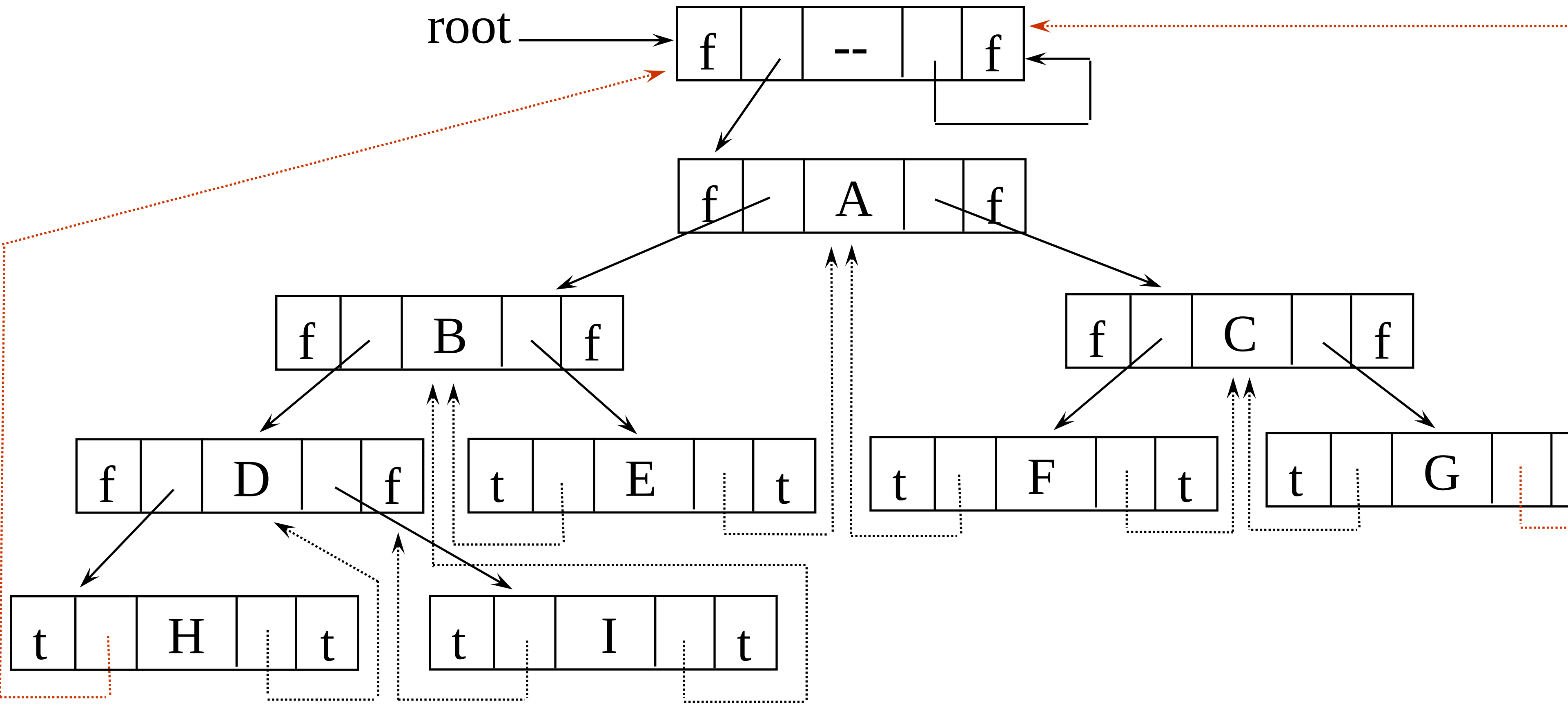


TRUE: thread

FALSE: child

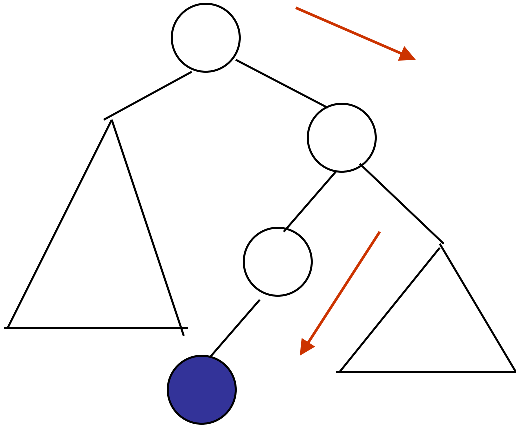
```
typedef struct threaded_tree
  *threaded_pointer;
typedef struct threaded_tree {
     short int left_thread;
     threaded_pointer left_child;
     char data;
     threaded_pointer right_child;
     short int right_thread;   };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer
  tree)
{
   threaded_pointer temp;
   temp = tree->right_child;
   if (!tree->right_thread)
     while (!temp->left_thread)
       temp = temp->left_child;
   return temp;
```



Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
/* traverse the threaded binary tree inorder */
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp==tree) break;
        printf("%3c", temp->data);
    }
```

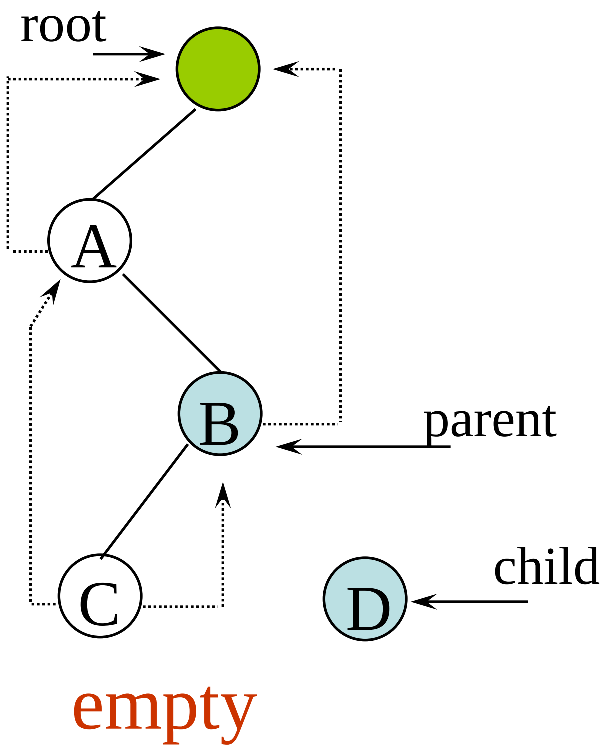
O(n)

Inserting Nodes into Threaded BTs

- Insert `child` as the right child of node `parent`
  - change `parent->right_thread` to FALSE
  - set `child->left_thread` and `child->right_threa`
    to TRUE
  - set `child->left_child` to point to `parent`
  - set `child->right_child` to `parent->right_child`
  - change `parent->right_child` to point to `child`

Examples

Insert a node D as a right child of B.



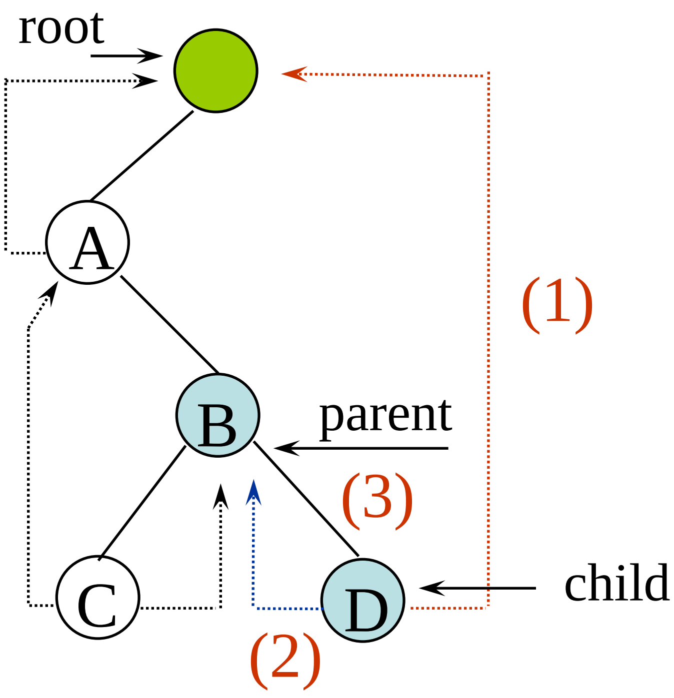
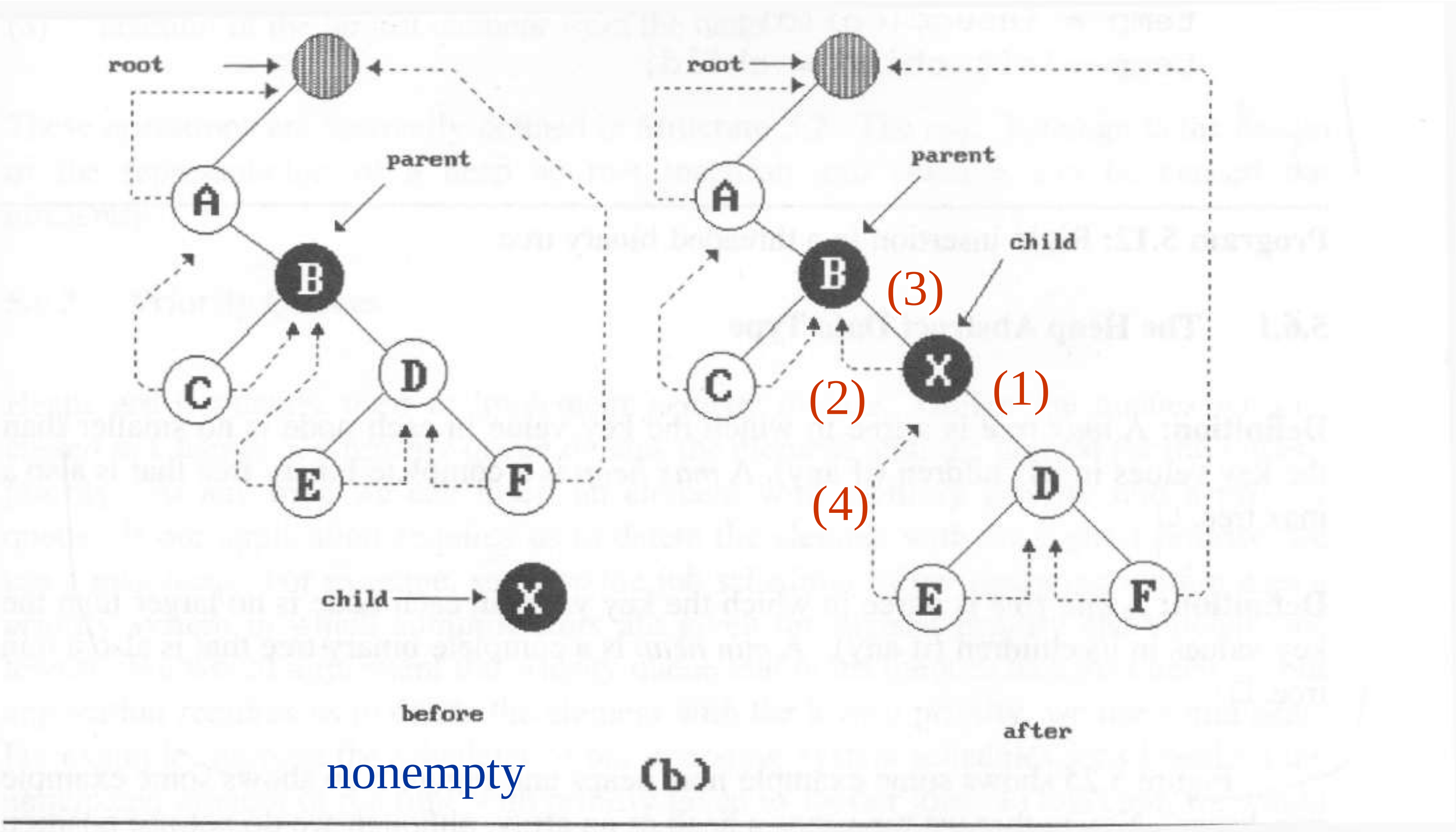


***Figure 5.24:** Insertion of child as a right child of parent in a threaded binary tree (p.217



Right Insertion in Threaded BTs

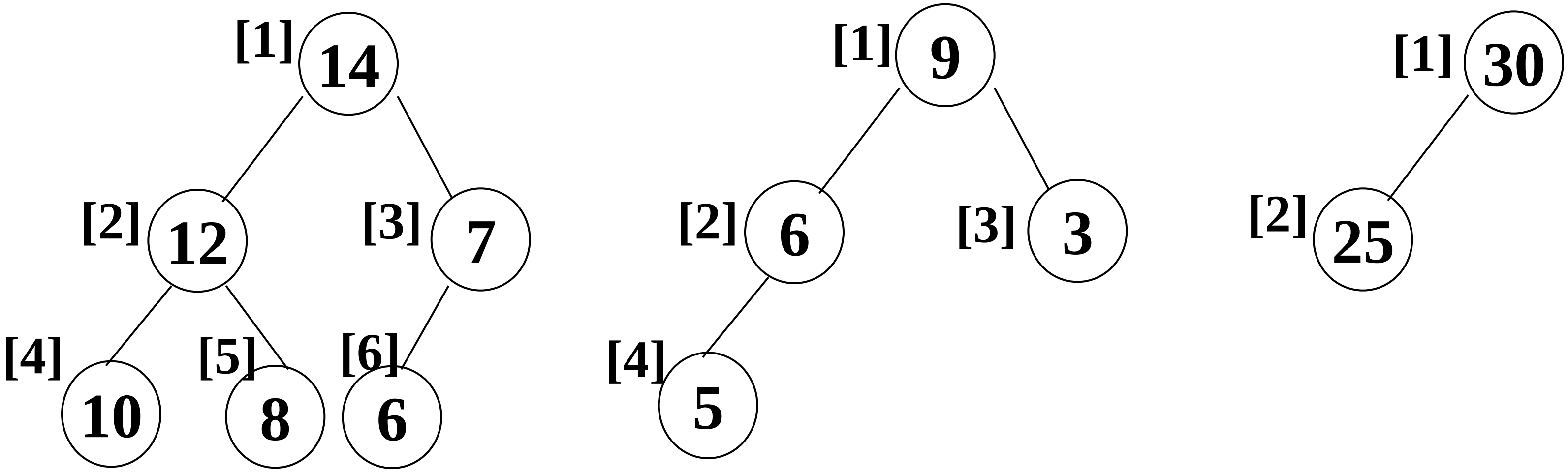
```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
     threaded_pointer temp;
(1)  child->right_child = parent->right_child;
     child->right_thread = parent->right_thread;
     child->left_child = parent;  case (a)
(2)  child->left_thread = TRUE;
(3)  parent->right_child = child;
     parent->right_thread = FALSE;
     if (!child->right_thread) { case (b)
(4)     temp = insucc(child);
        temp->left_child = child;
     }
}
```

Lecture 5

Heap

- A *max tree* is a tree in which the key value i each node is no smaller than the key values its children. A *max heap* is a complete bina tree that is also a max tree.
- A *min tree* is a tree in which the key value ir each node is no larger than the key values i its children. A *min heap* is a complete binar tree that is also a min tree.
- Operations on heaps
  - creation of an empty heap
  - insertion of a new element into the heap;

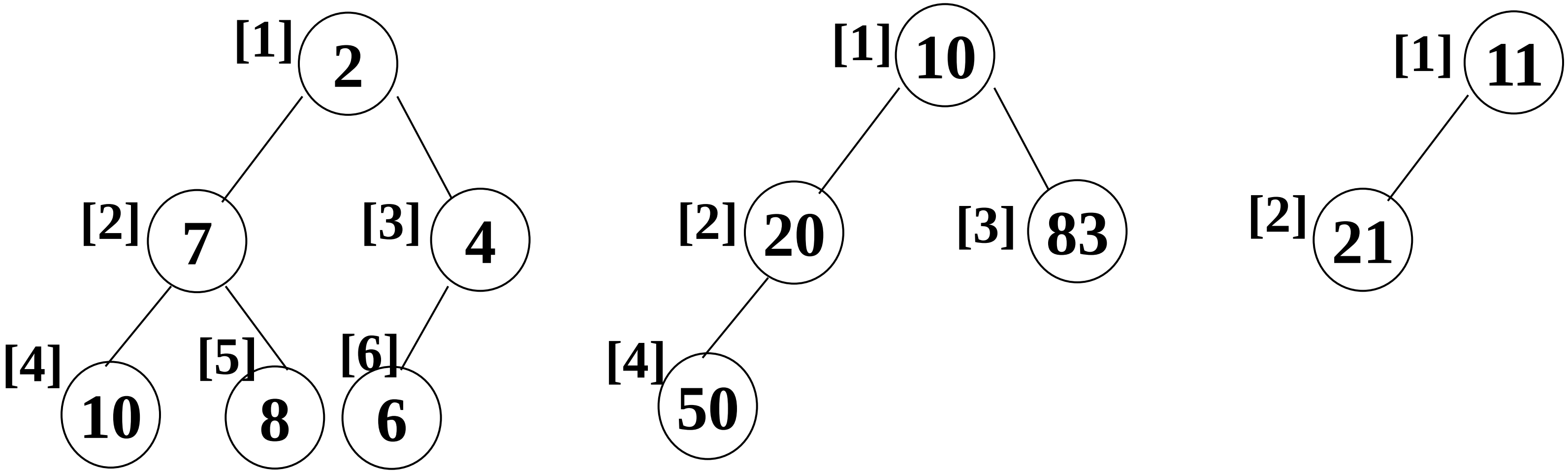
***Figure 5.25:** Sample max heaps (p.219)



Property:

The root of max heap (min heap) contains the largest (smallest).

***Figure 5.26:**Sample min heaps (p.220)



ADT for Max Heap

structure MaxHeap

objects: a complete binary tree of n > 0 elements organized s
that
the value in each node is at least as large as those in its
children

functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*,
max_size belong to integer

MaxHeap Create(max_size)::= create an empty heap that c
hold a maximum of max_size elements

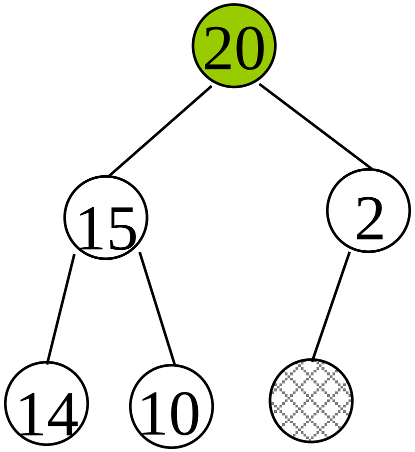
Boolean HeapFull(heap, n)::= if (n==max_size) return TRU
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n)) inse
item into heap and return the resulting h
else return error

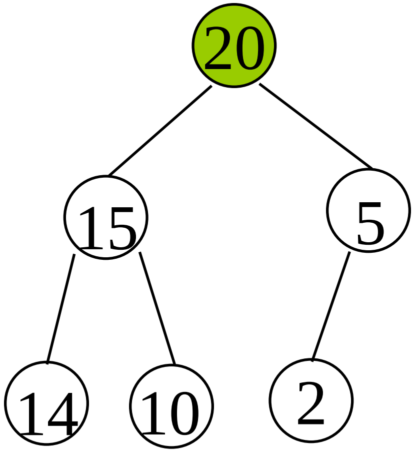
Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return
instance of the largest element in the he
and remove it from the heap

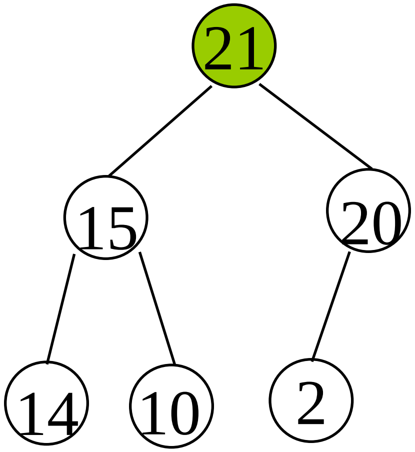
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

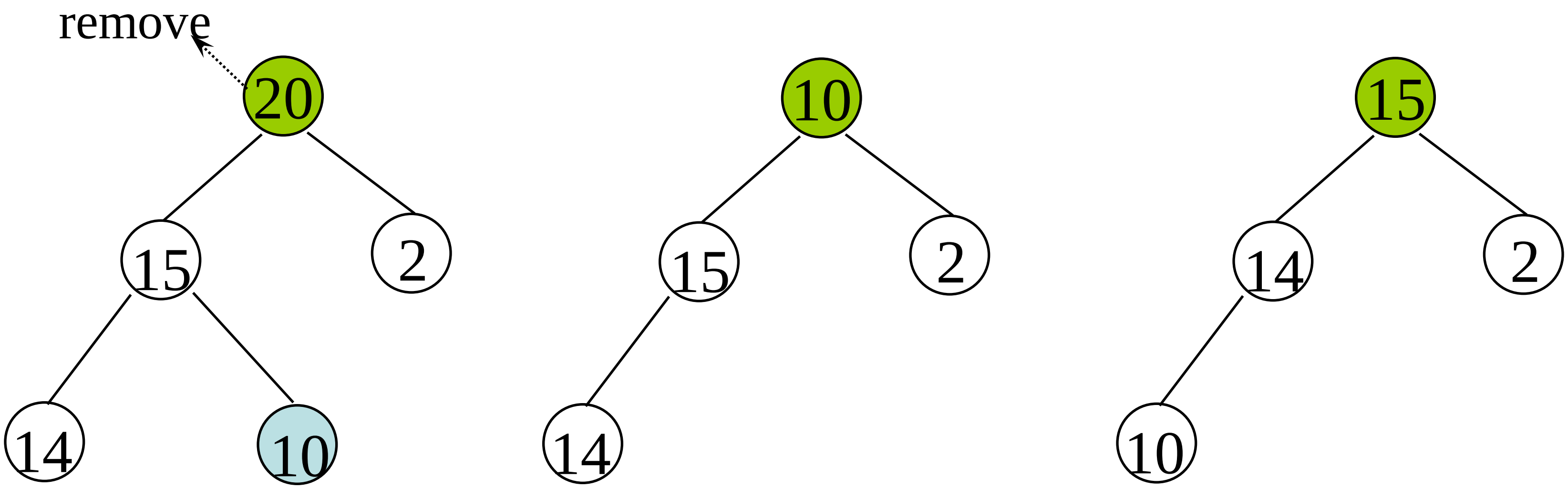
Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, “the heap is full.\n”);
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

$2^k-1=n \implies k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

Example of Deletion from Max Heap



Deletion from a Max Heap

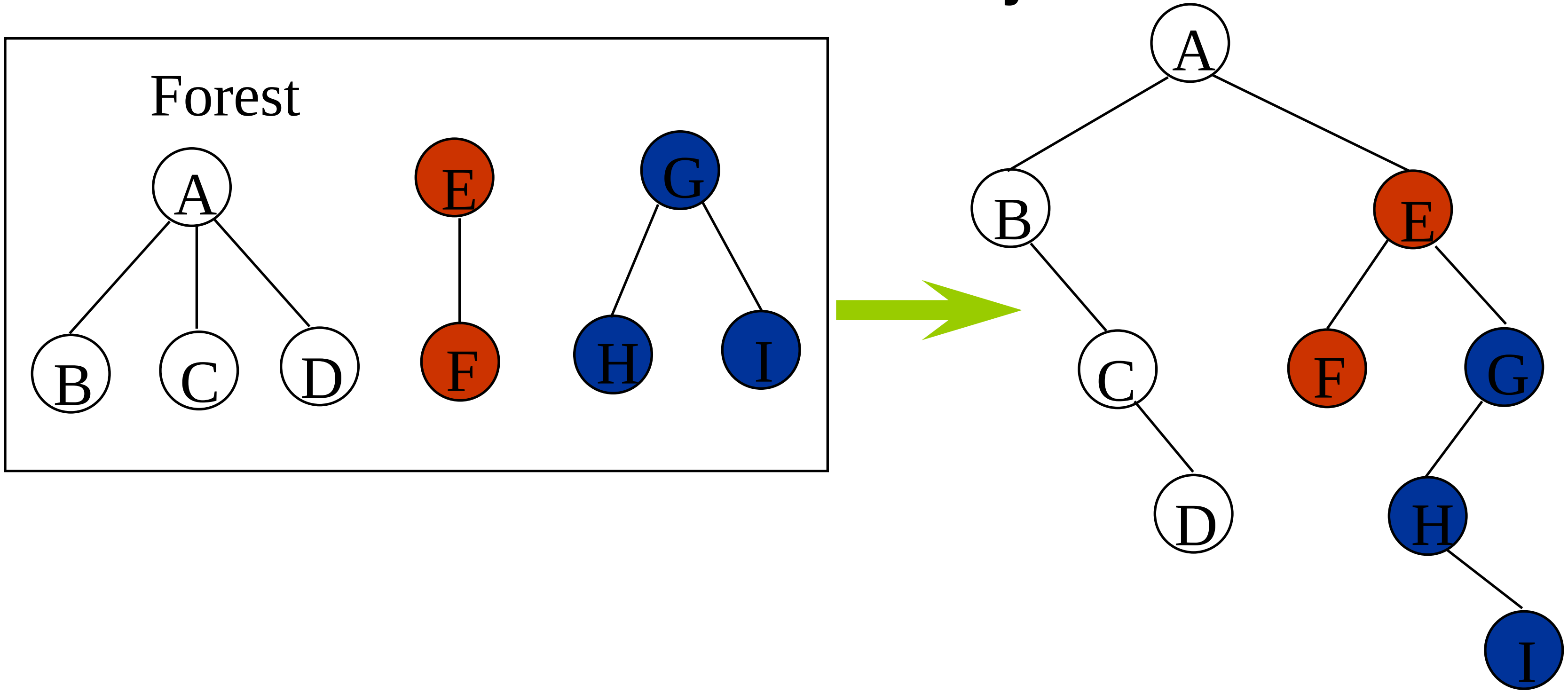
```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
```

```
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
         (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

Lecture 6

Forest

- A forest is a set of n >= 0 disjoint trees



Transform a forest into a binary tree

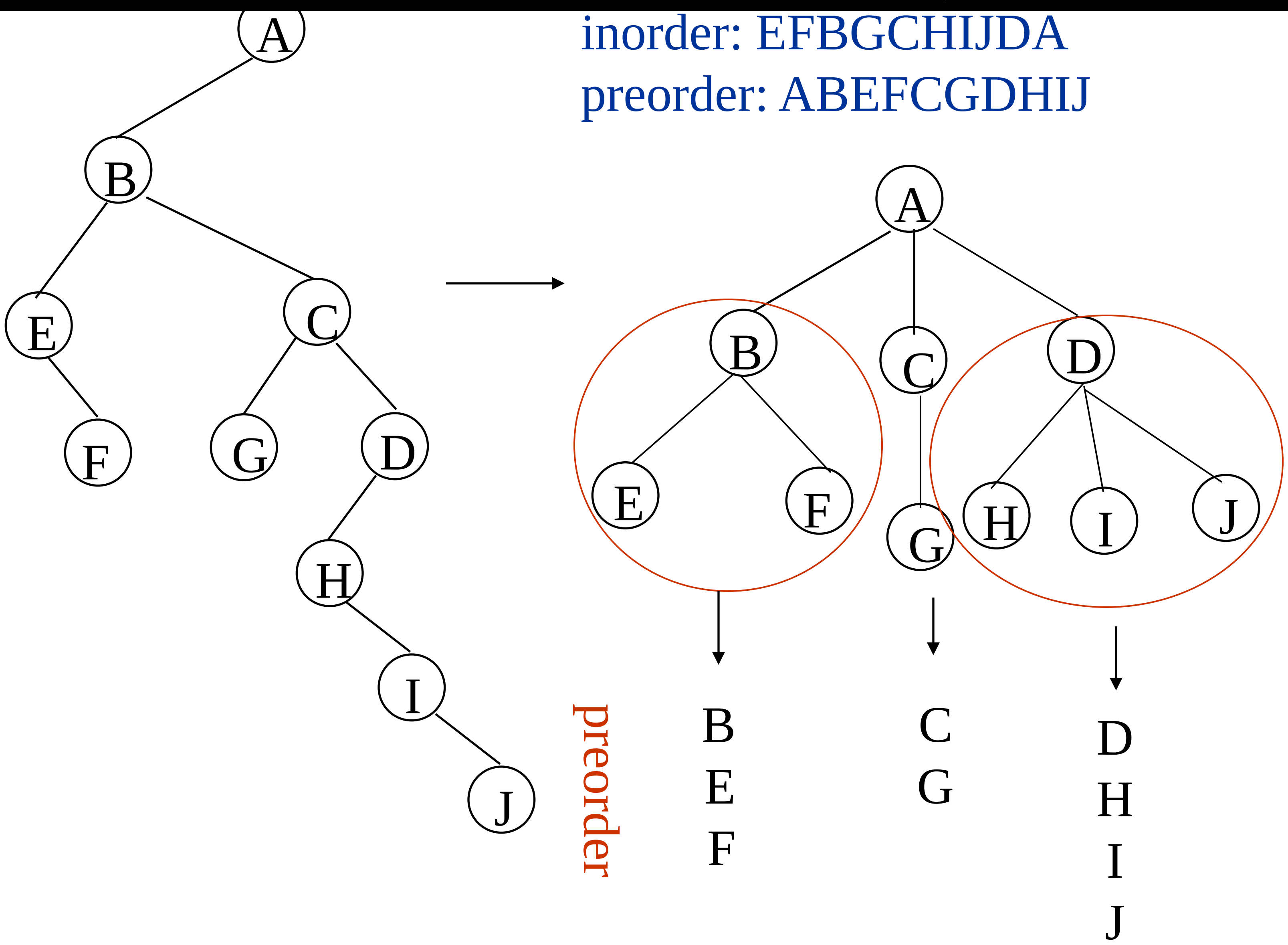
- T1, T2, …, T*n*: a forest of trees
  B(T1, T2, …, Tn): a binary tree corresponding to this forest
- algorithm
  (1) empty, if n = 0
  (2) has root equal to root(T1)
  has left subtree equal to B(T11,T12,…,T1*m*)
  has right subtree equal to B(T2,T3,…,Tn)

Forest Traversals

- Preorder
  - If F is empty, then return
  - Visit the root of the first tree of F
  - Taverse the subtrees of the first tree in tree pre
  - Traverse the remaining trees of F in preorder
- Inorder
  - If F is empty, then return
  - Traverse the subtrees of the first tree in tree in
  - Visit the root of the first tree
  - Traverse the remaining trees of F is indorer

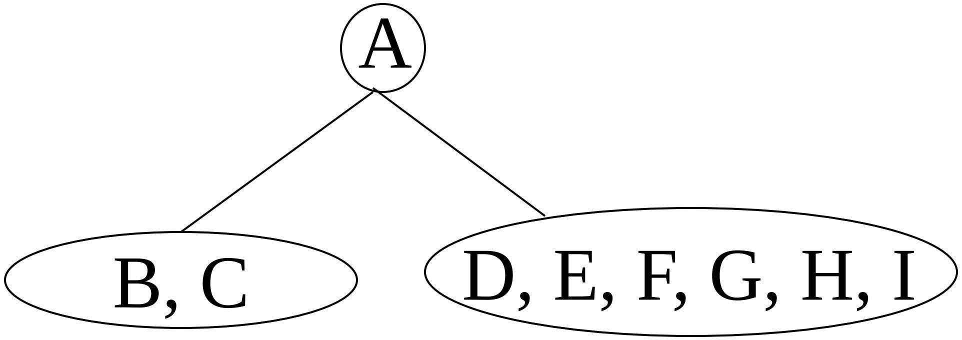
inorder: EFBGCHIJDA

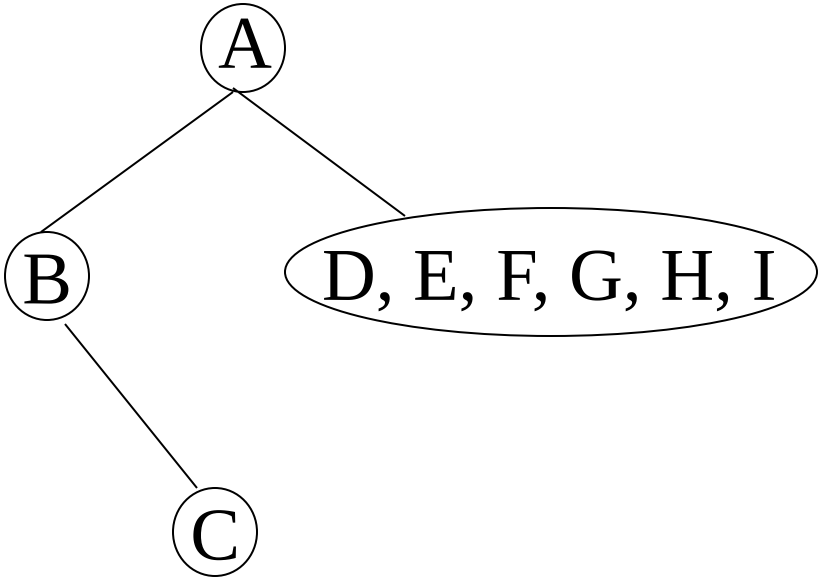
preorder: ABEFCGDHIJ

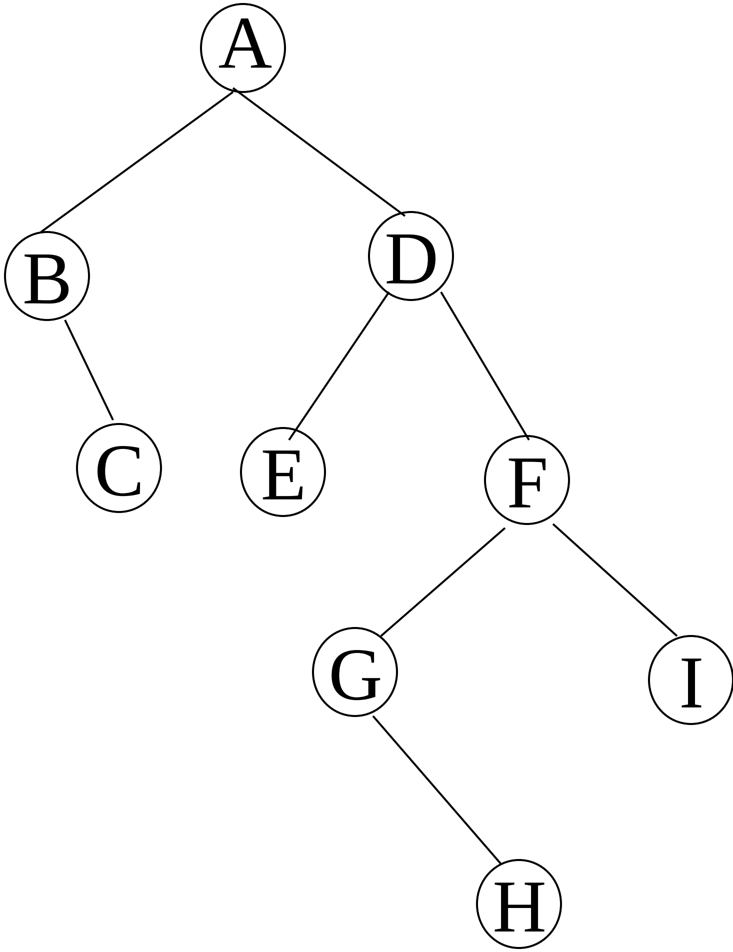


preorder: A B C D E F G H I

inorder: B C A E D G H F I







Lecture 7-8

AVL (Height-balanced Trees)

- A **perfectly balanced** binary tree is a binary tree such that:
  - The height of the left and right subtrees of the root are equal
  - The left and right subtrees of the root are perfectly balanced binary trees

Perfectly Balanced Binary Tree

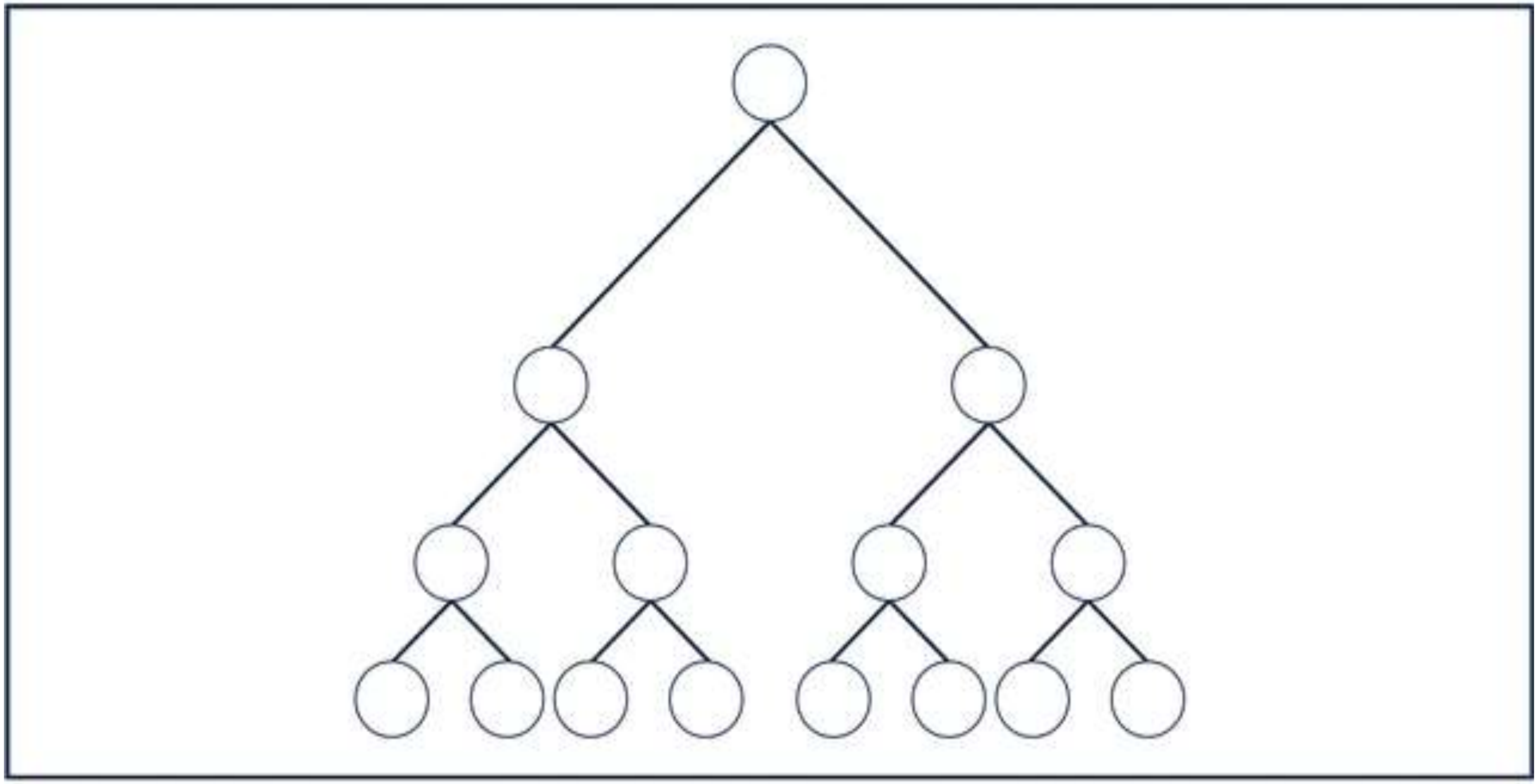


Figure 11-18 Perfectly balanced binary tree

AVL (Height-balanced Trees)

- An **AVL tree** (or **height-balanced tree**) is a binary search tree such that:
    - – The height of the left and right subtrees of the root differ by at most 1
    - – The left and right subtrees of the root are AVL trees
    - – Node balance factor of -1 if node left high, 0 if node is equal high and +1 is node is right high

AVL Trees

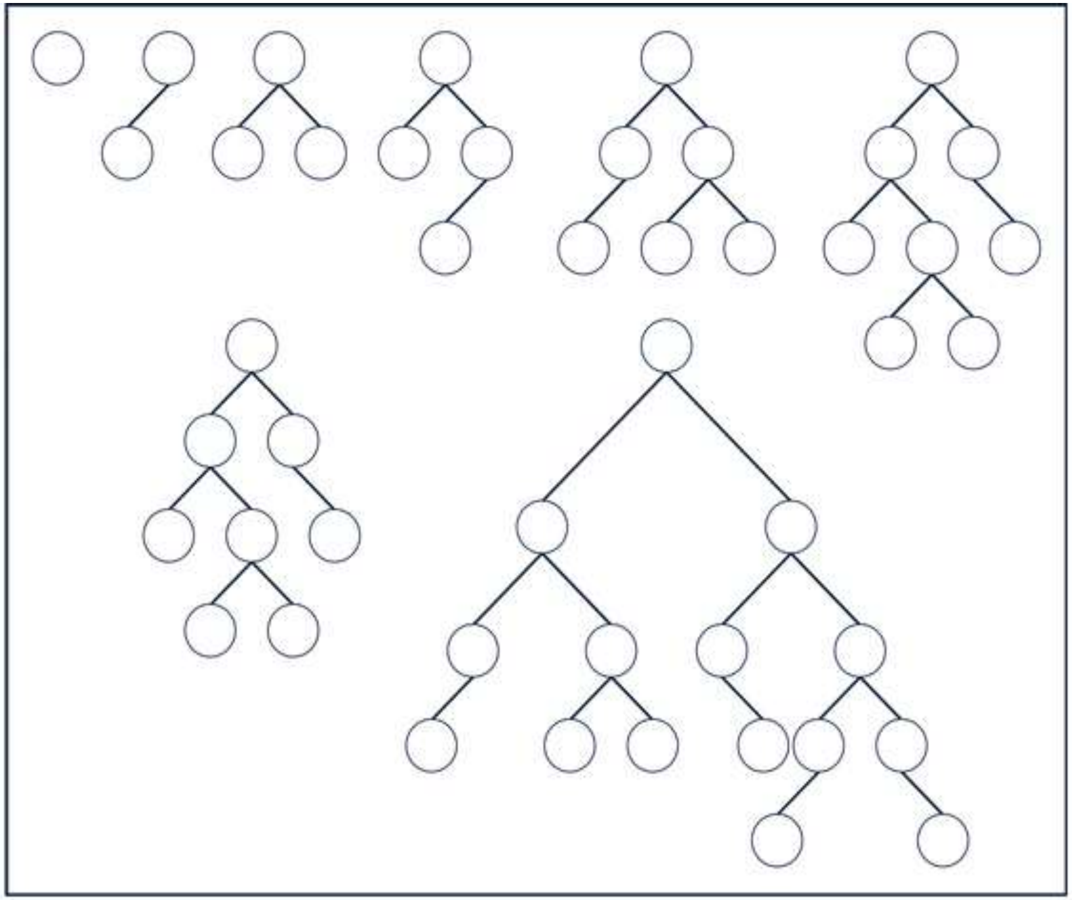


Figure 11-19 AVL trees

Non-AVL Trees

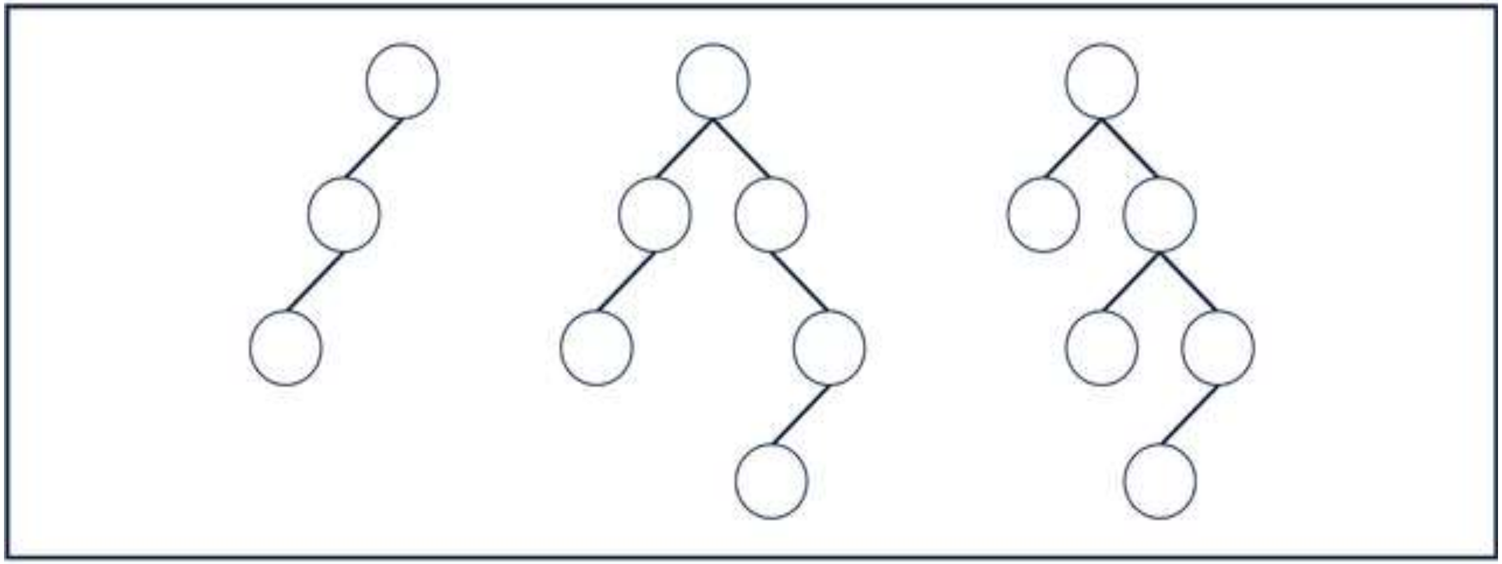


Figure 11-20 Non-AVL trees

Insertion Into AVL Tree

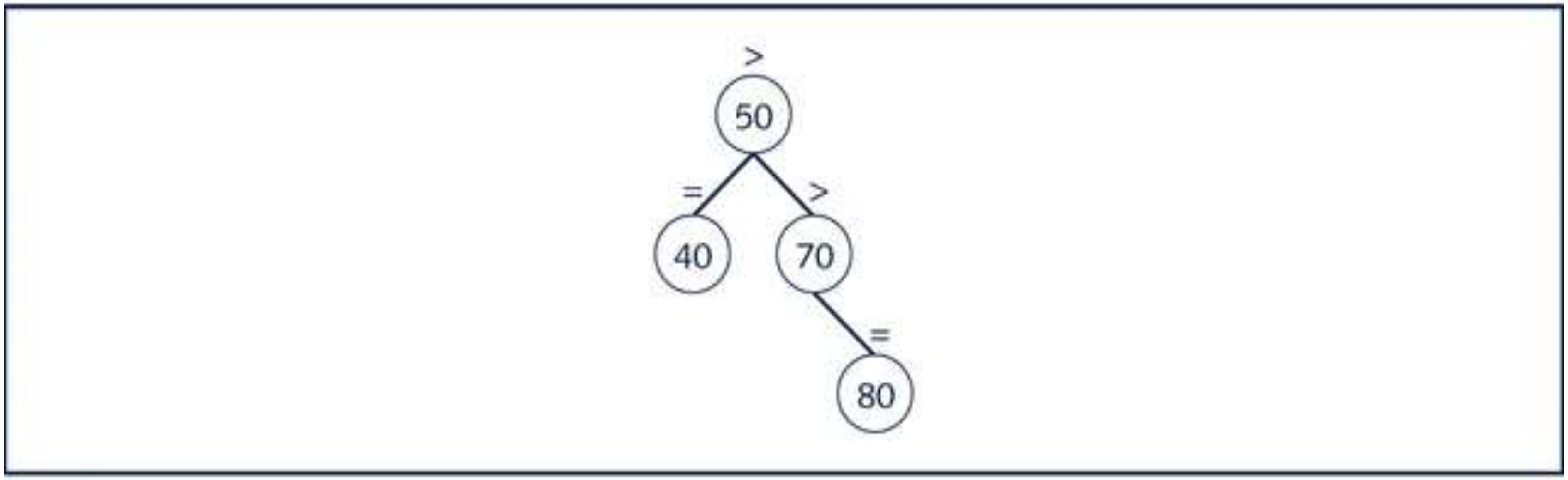


Figure 11-21 AVL tree before inserting 90

Insertion Into AVL Trees

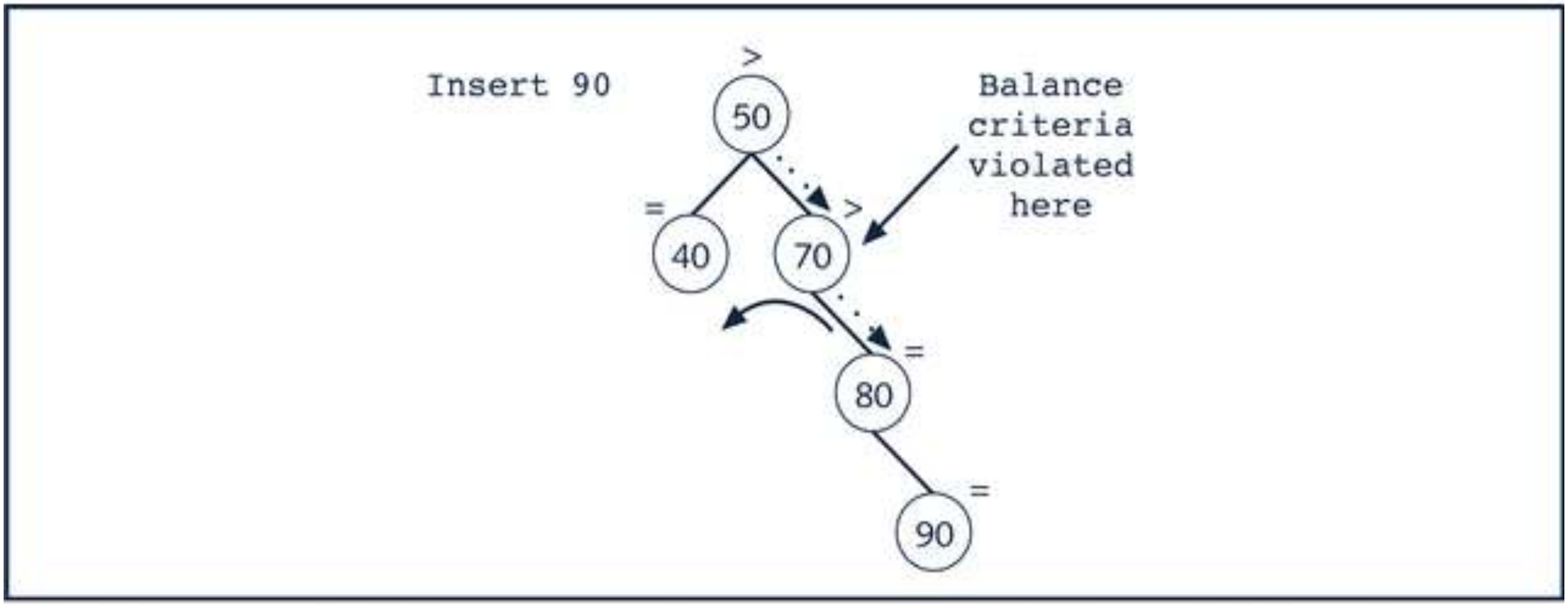


Figure 11-22 Binary tree of Figure 11-21 after inserting 90; nodes other than 90 show their balance factors before insertion

Insertion Into AVL Trees

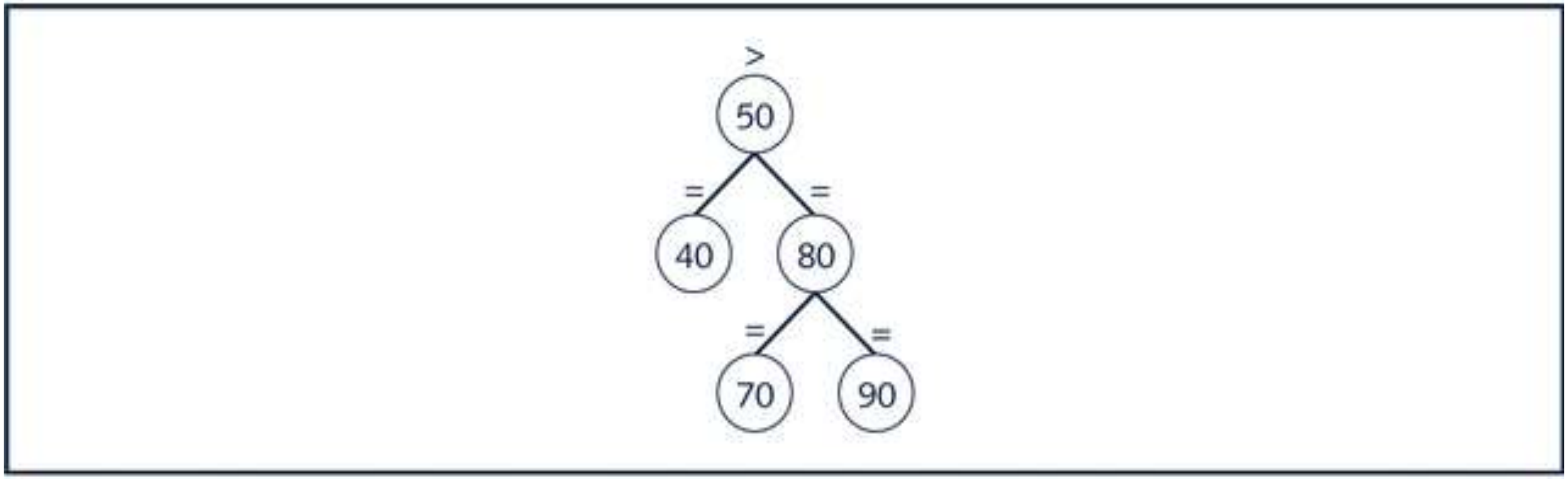


Figure 11-23 AVL tree of Figure 11-21 after inserting 90 and adjusting the balance factors

Insertion Into AVL Trees

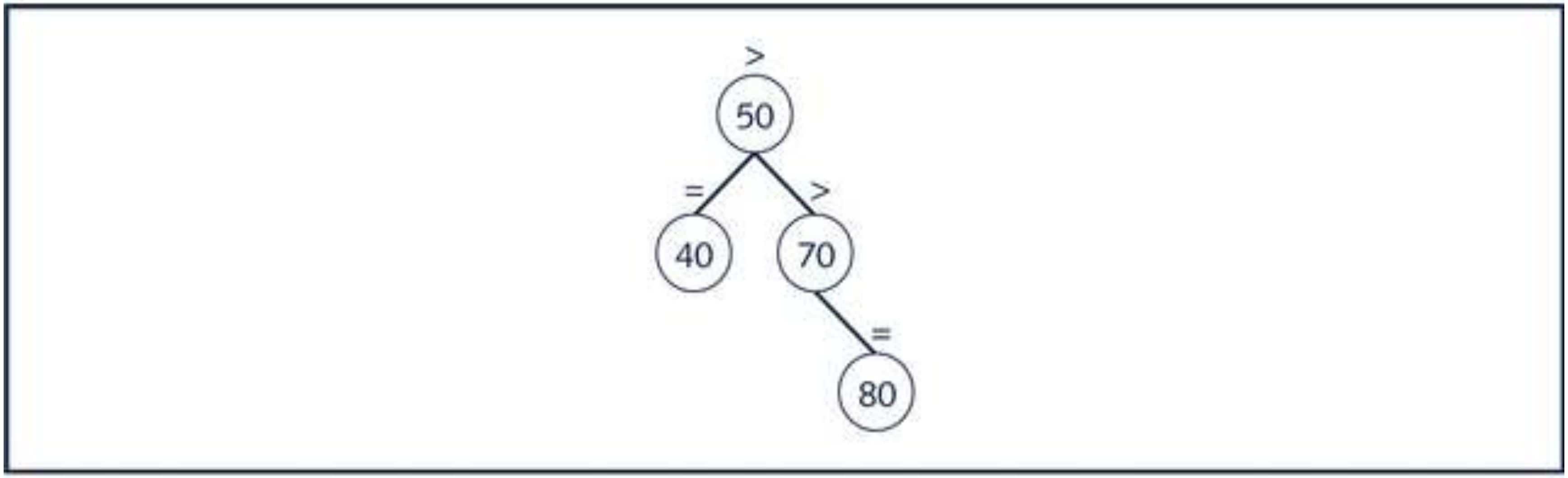


Figure 11-24 AVL tree before inserting 75

Insertion Into AVL Trees

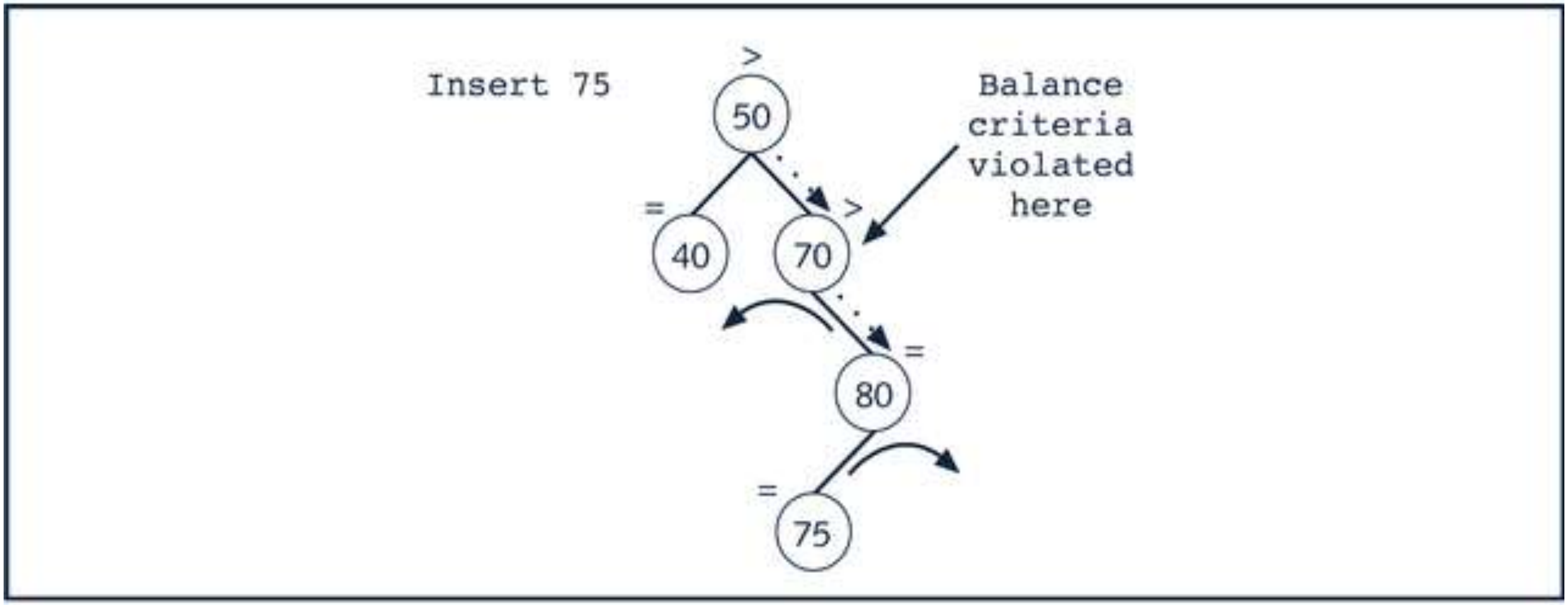


Figure 11-25 Binary tree of Figure 11-24 after inserting 75; nodes other than 75 show their balance factors before insertion

Insertion Into AVL Trees

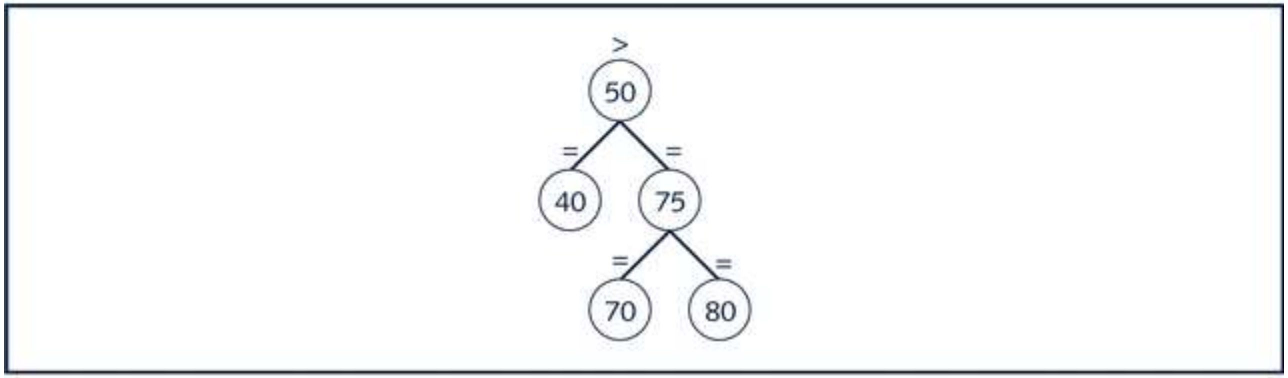


Figure 11-26 AVL tree of Figure 11-24 after inserting 75 and adjusting the balance factors

Insertion Into AVL Trees

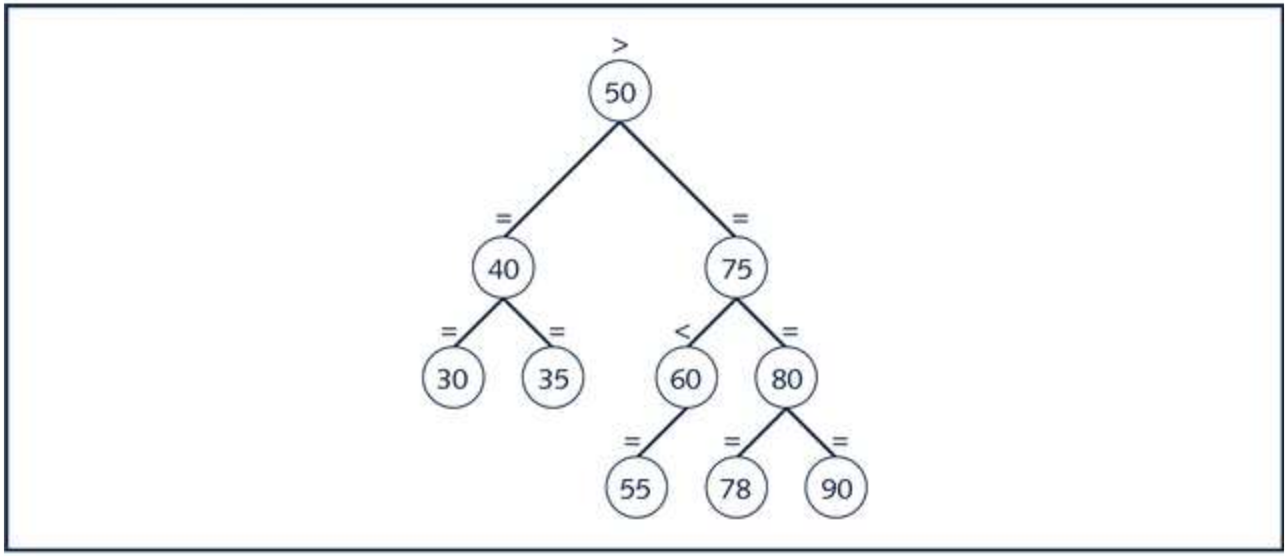


Figure 11-27 AVL tree before inserting 95

Insertion Into AVL Trees

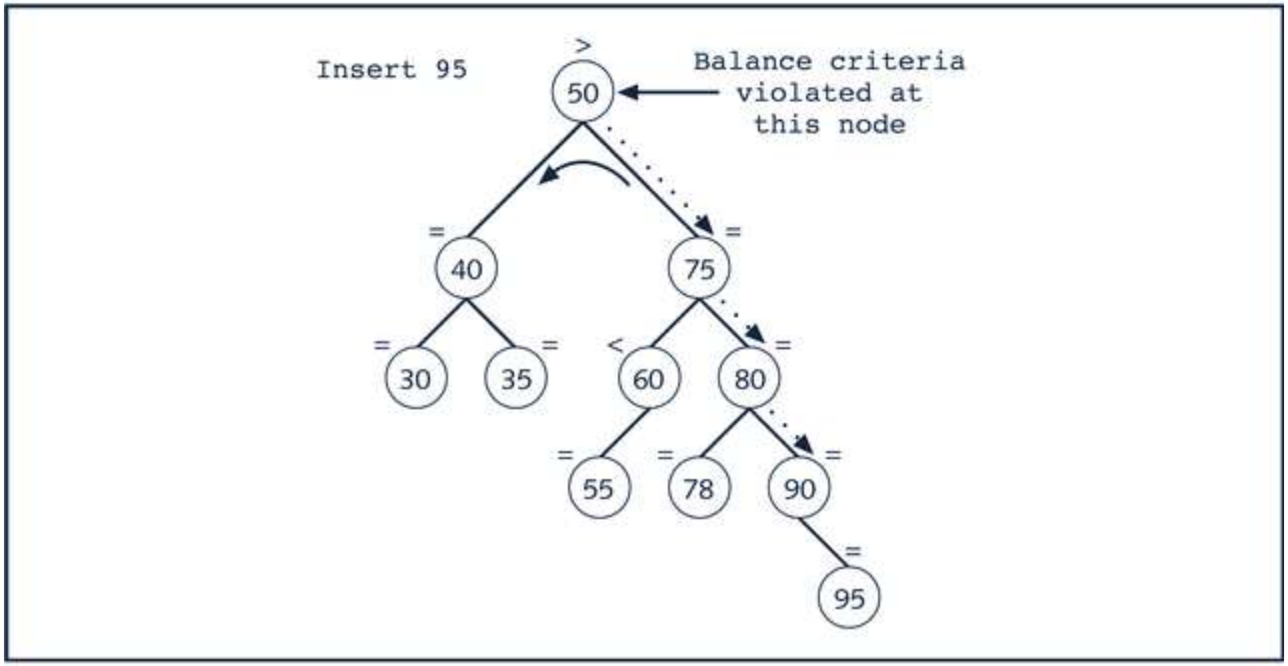


Figure 11-28 Binary tree of Figure 11-27 after inserting 95; nodes other than 95 show their balance factors before insertion

Insertion Into AVL Trees

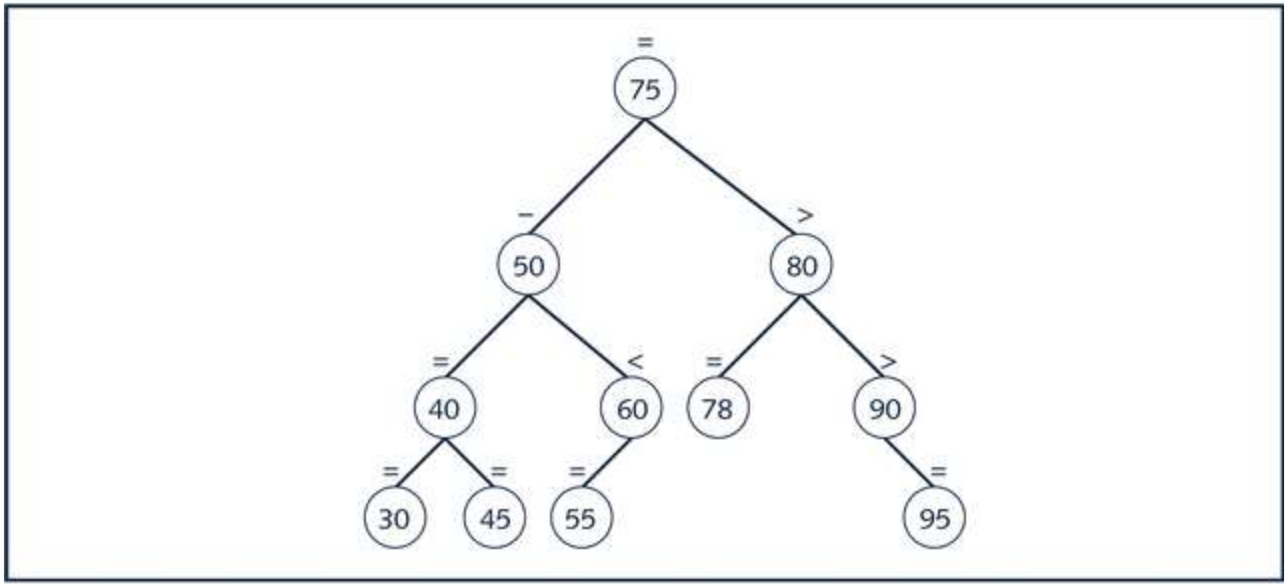


Figure 11-29 AVL tree of Figure 11-27 after inserting 95 and adjusting the balance factors

Insertion Into AVL Trees

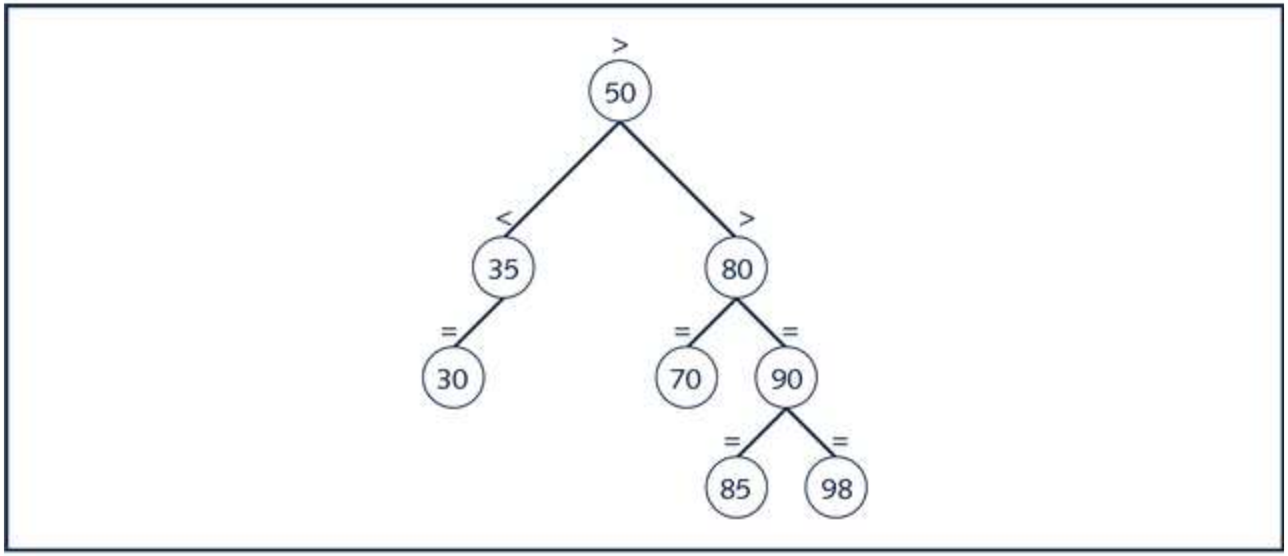


Figure 11-30 AVL tree before inserting 88

Insertion Into AVL Trees

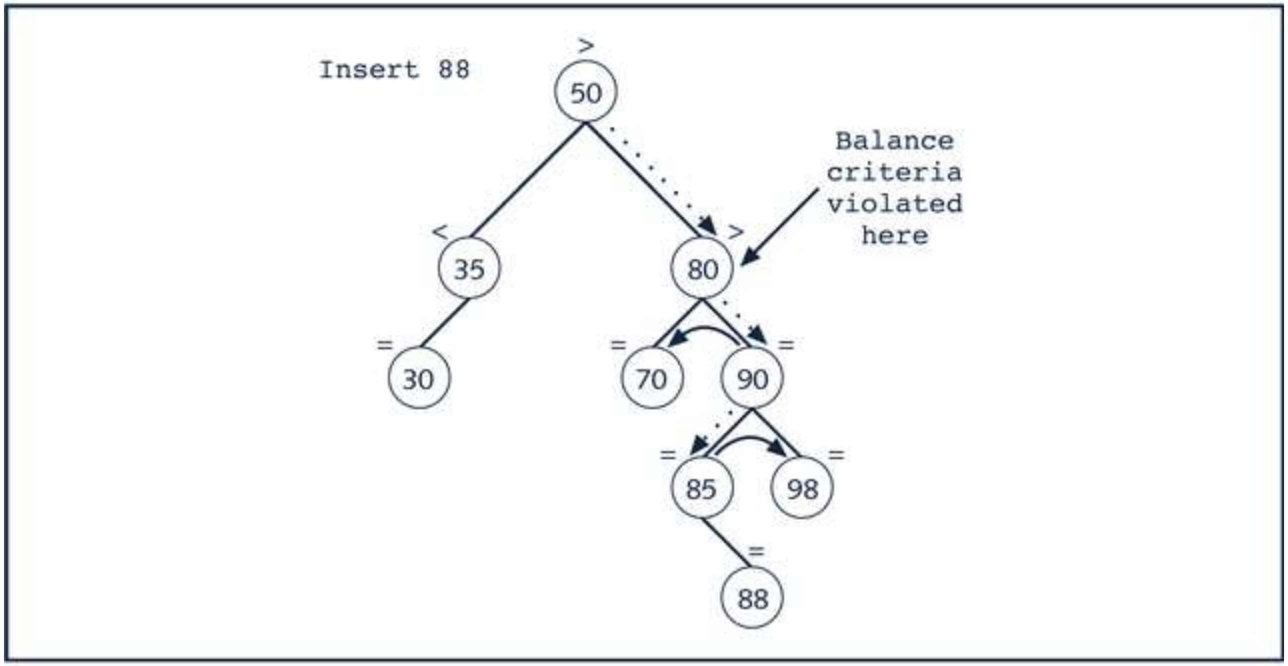


Figure 11-31 Binary tree of Figure 11-30 after inserting 88; nodes other than 88 show their balance factors before insertion

Insertion Into AVL Trees

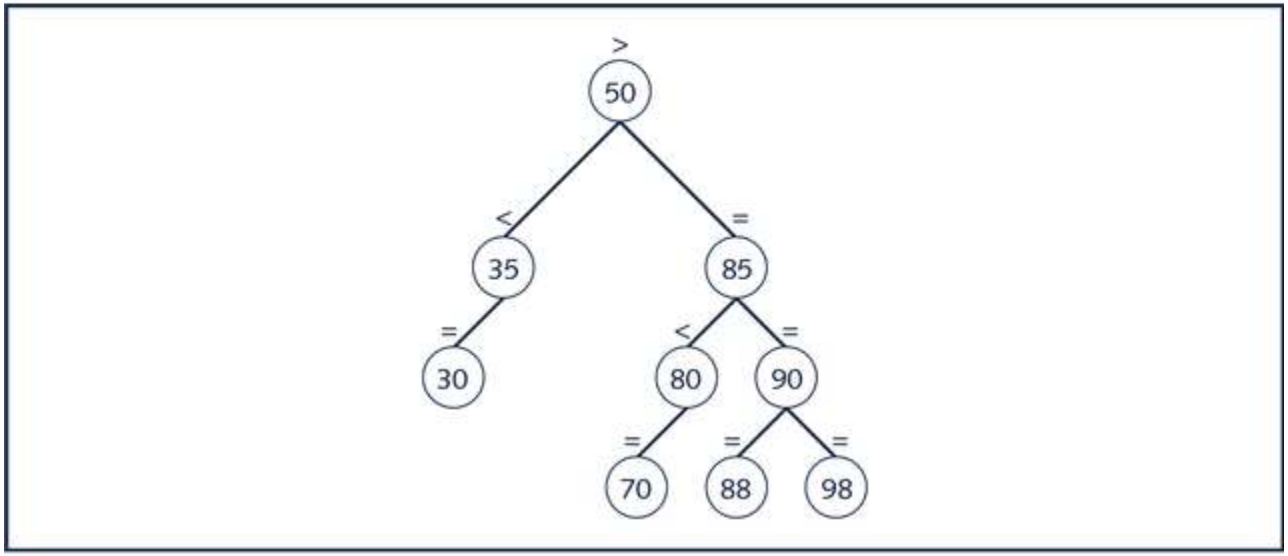


Figure 11-32 AVL tree of Figure 11-30 after inserting 88 and adjusting the balance factors

AVL Tree Rotations

- Reconstruction procedure: **rotating** tree
- **left rotation** and **right rotation**
- Suppose that the rotation occurs at node *x*
- Left rotation: certain nodes from the right subtree of *x* move to its left subtree; the root of the right subtree of *x* becomes the new root of the reconstructed subtree
- Right rotation at *x*: certain nodes from the left subtree of *x* move to its right subtree; the root of the left subtree of *x* becomes the new root of the reconstructed subtree

AVL Tree Rotations

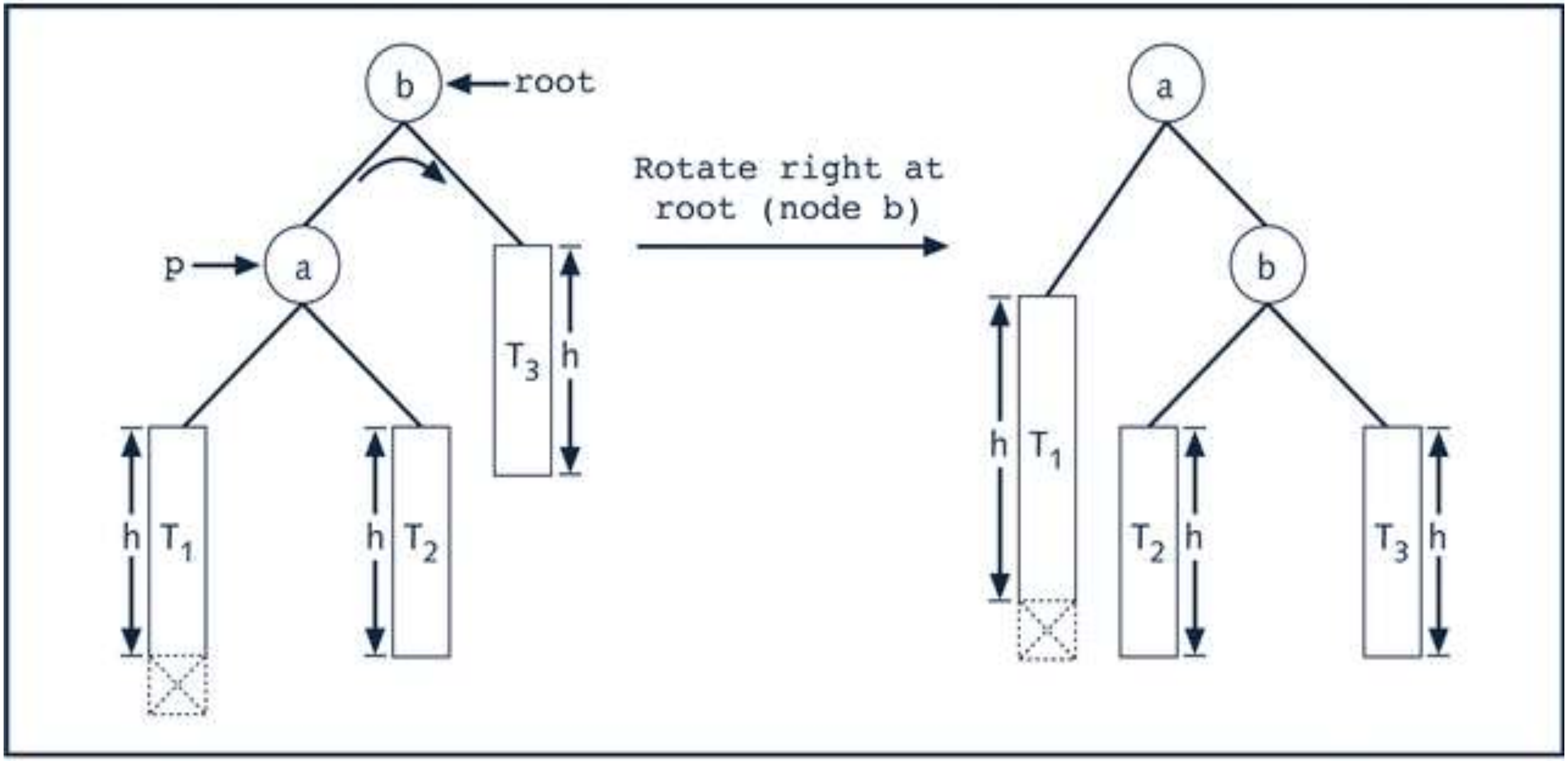


Figure 11-33 Right rotation at *b*

AVL Tree Rotations

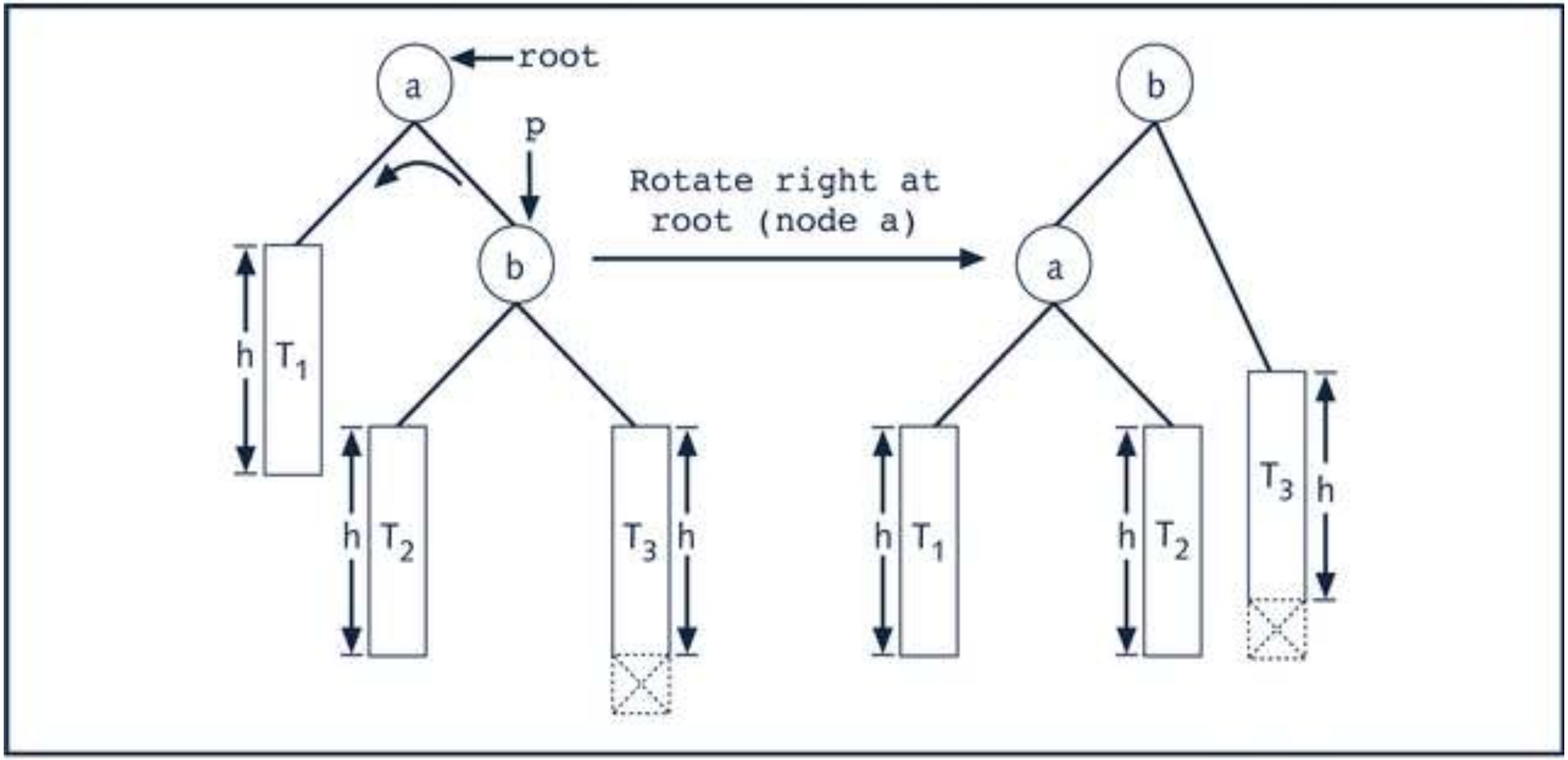


Figure 11-34 Left rotation at *a*

AVL Tree Rotations

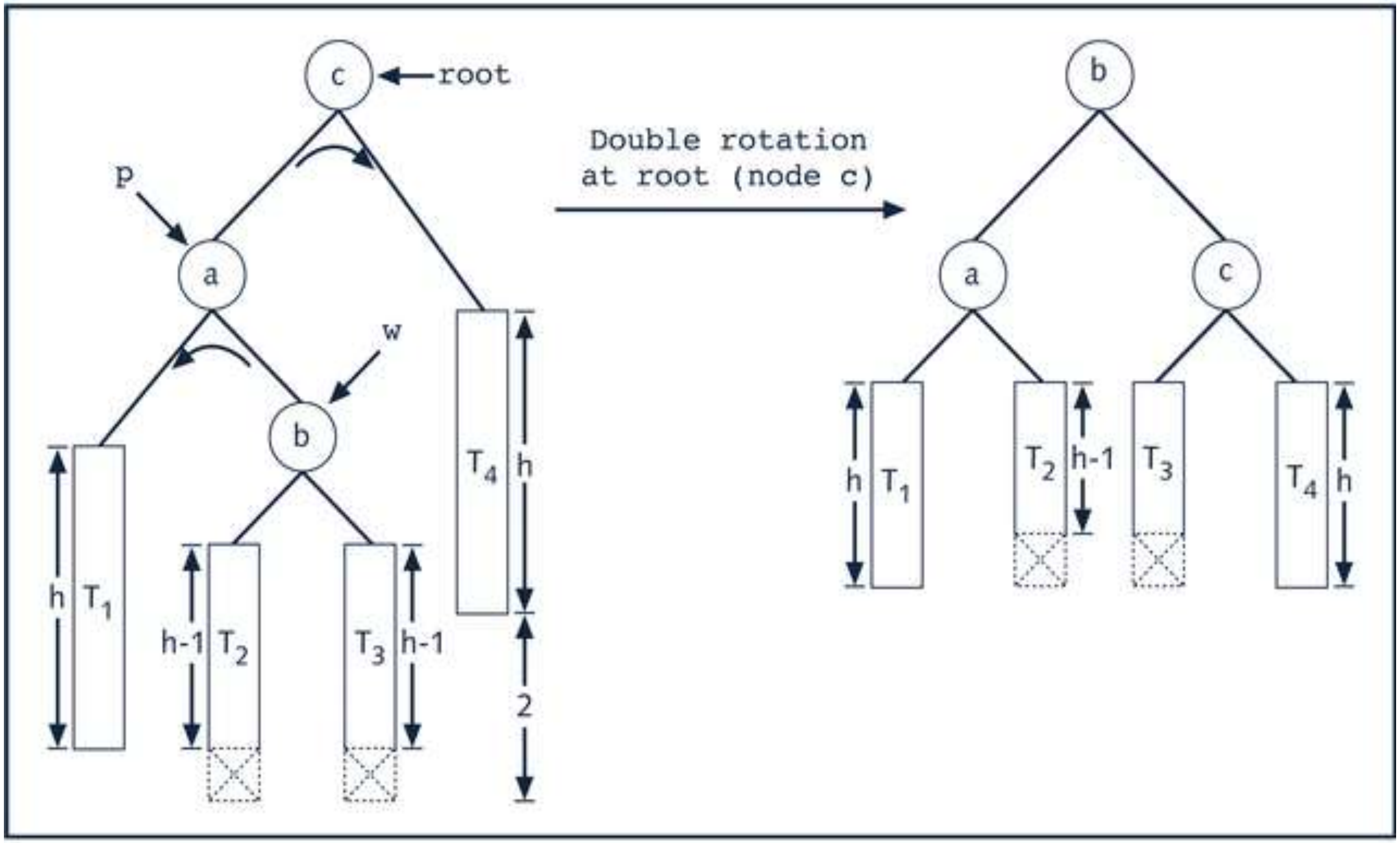


Figure 11-35 Double rotation: first rotate left at *a*, then rotate right at *c*

AVL Tree Rotations

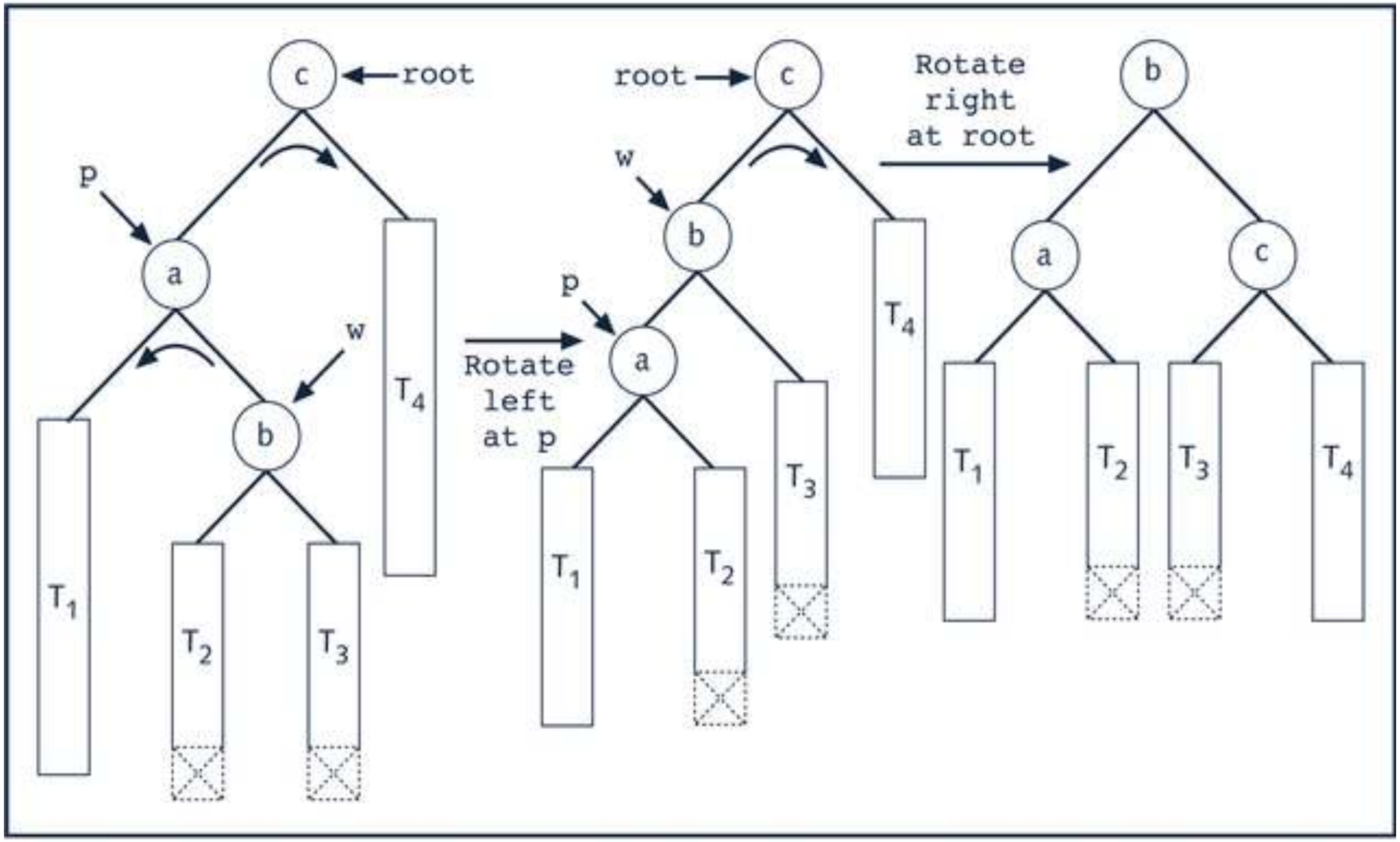


Figure 11-36 Left rotation at *a* followed by a right rotation at *c*

AVL Tree Rotations

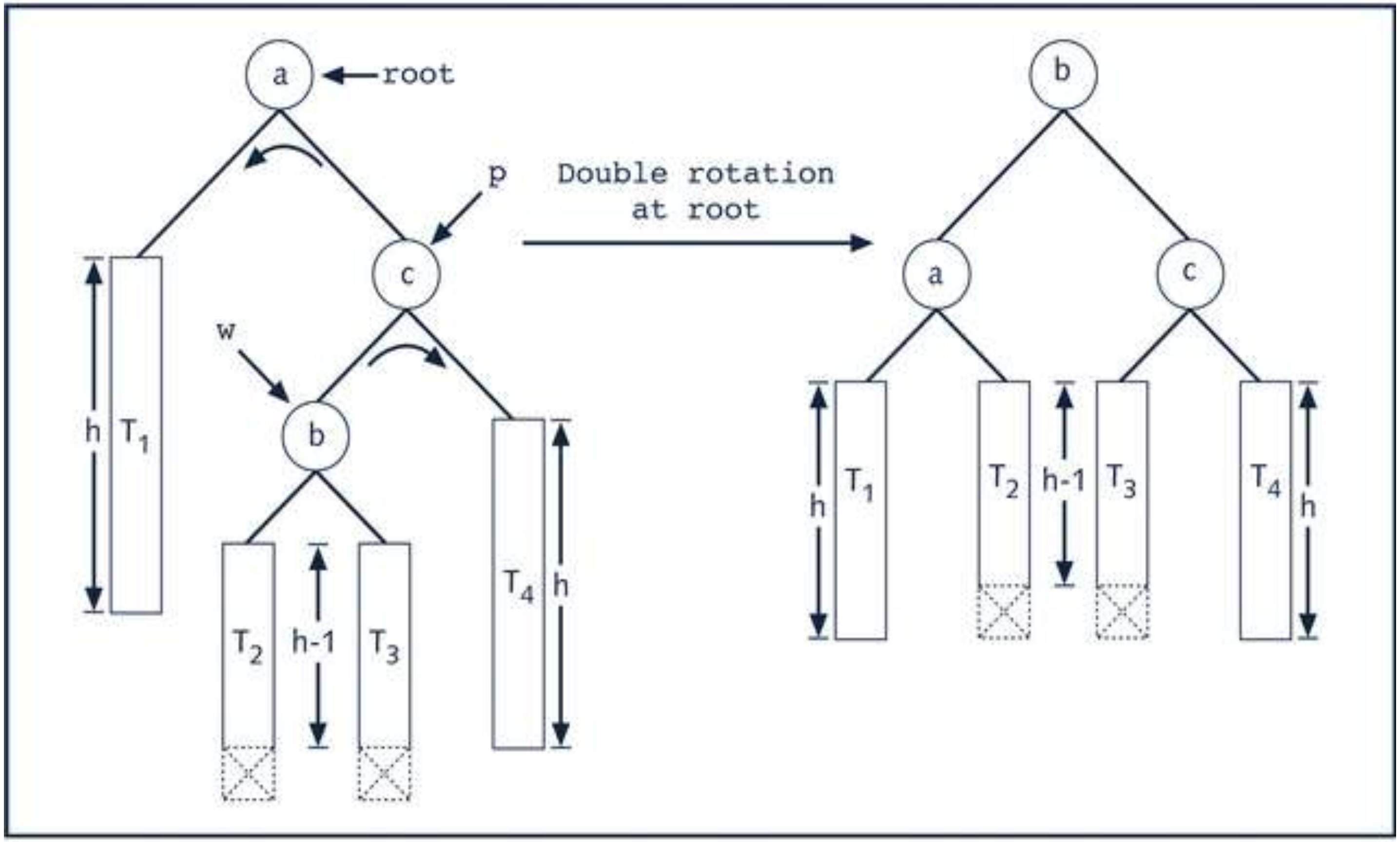


Figure 11-37 Double rotation: first rotate right at *c*, then rotate left at *a*

AVL Tree Rotations

Figure 11-38 AVL tree after inserting 40

AVL Tree Rotations

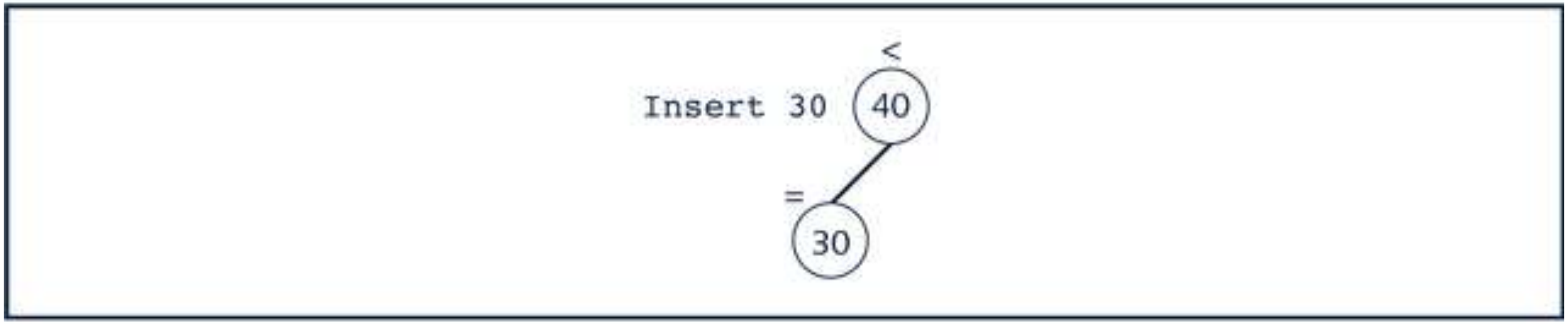


Figure 11-39 AVL tree after inserting 30

AVL Tree Rotations

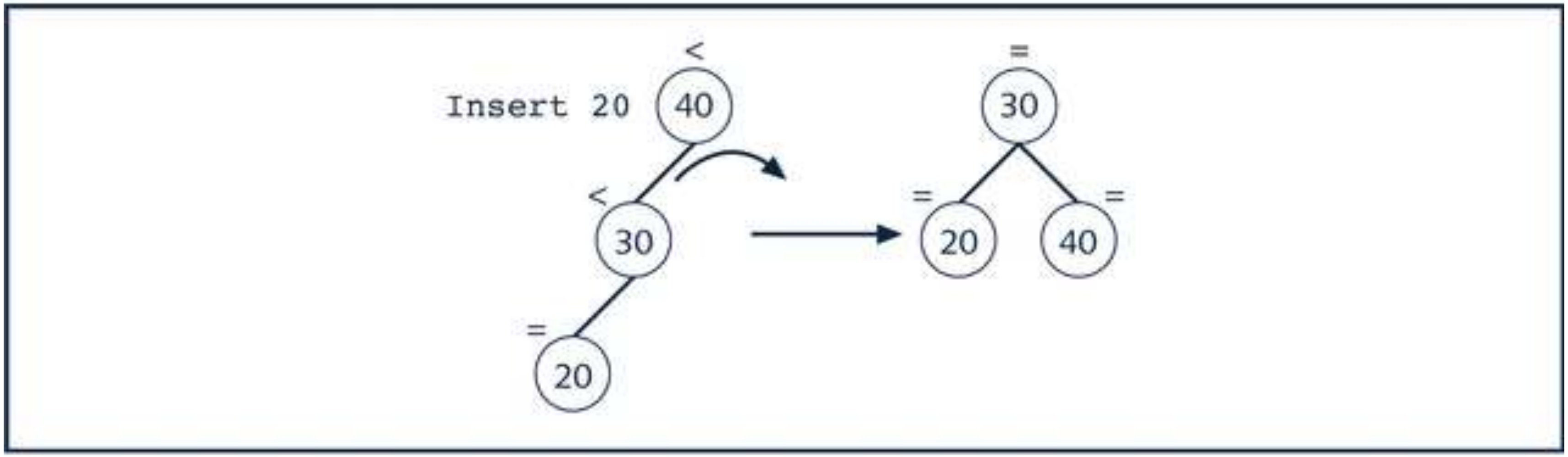


Figure 11-40 AVL tree after inserting 20

AVL Tree Rotations

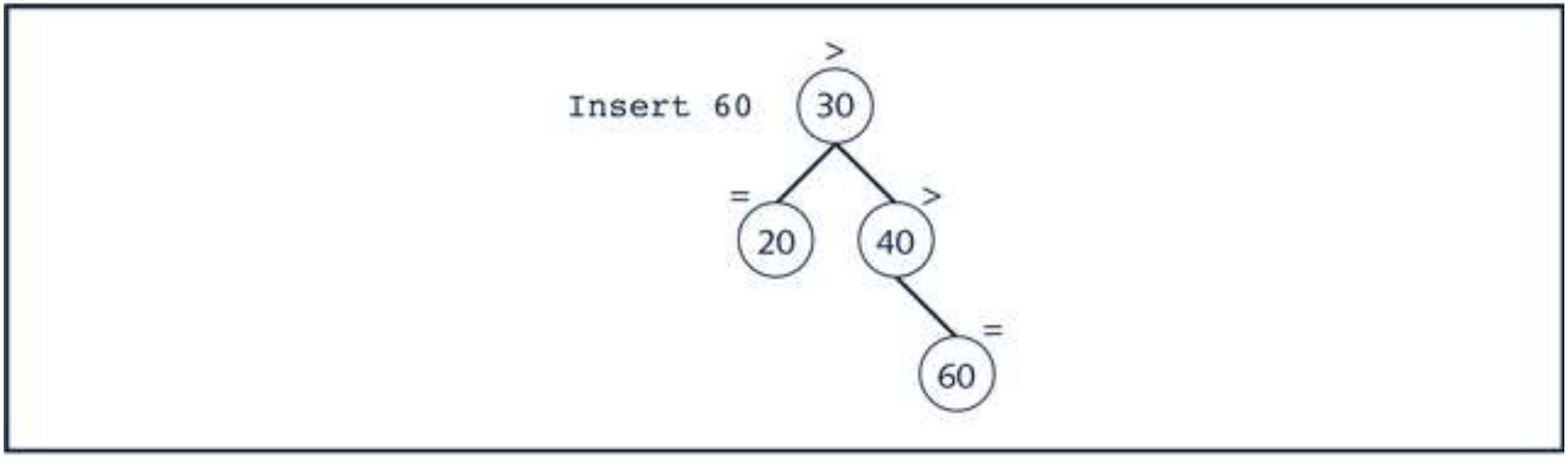


Figure 11-41 AVL tree after inserting 60

AVL Tree Rotations

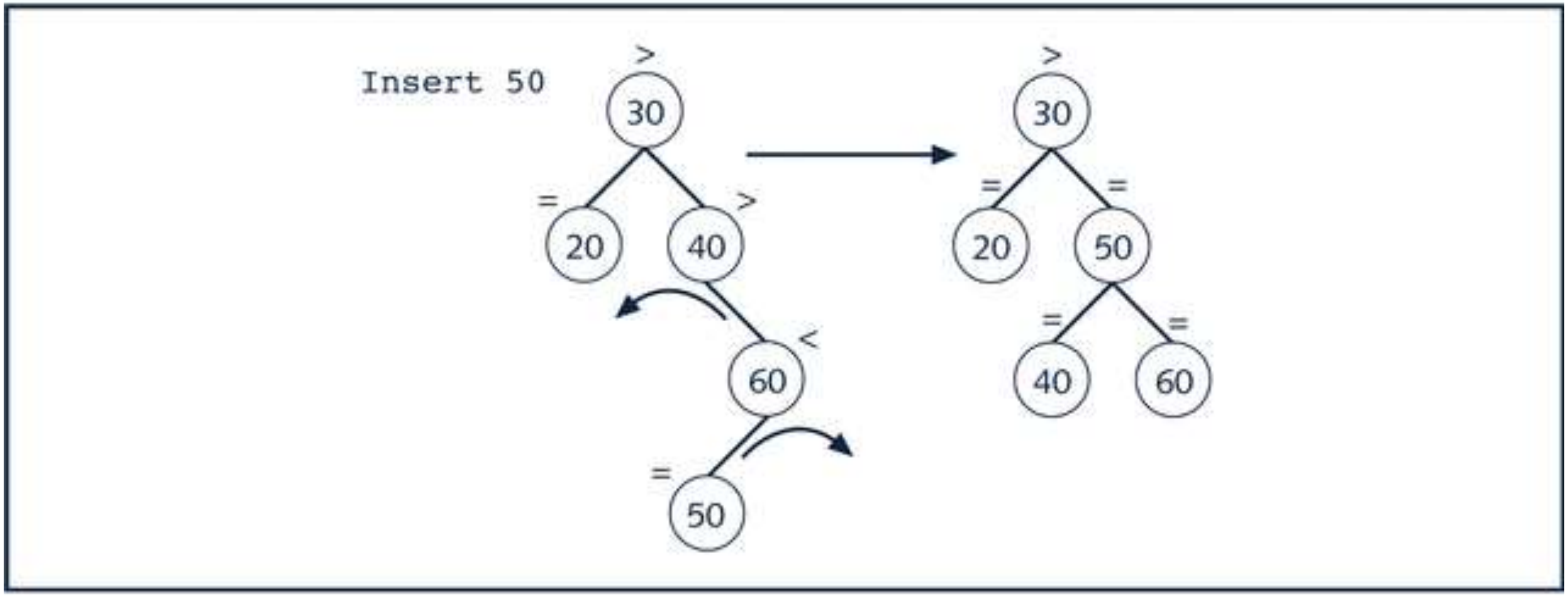


Figure 11-42 AVL tree after inserting 50

AVL Tree Rotations

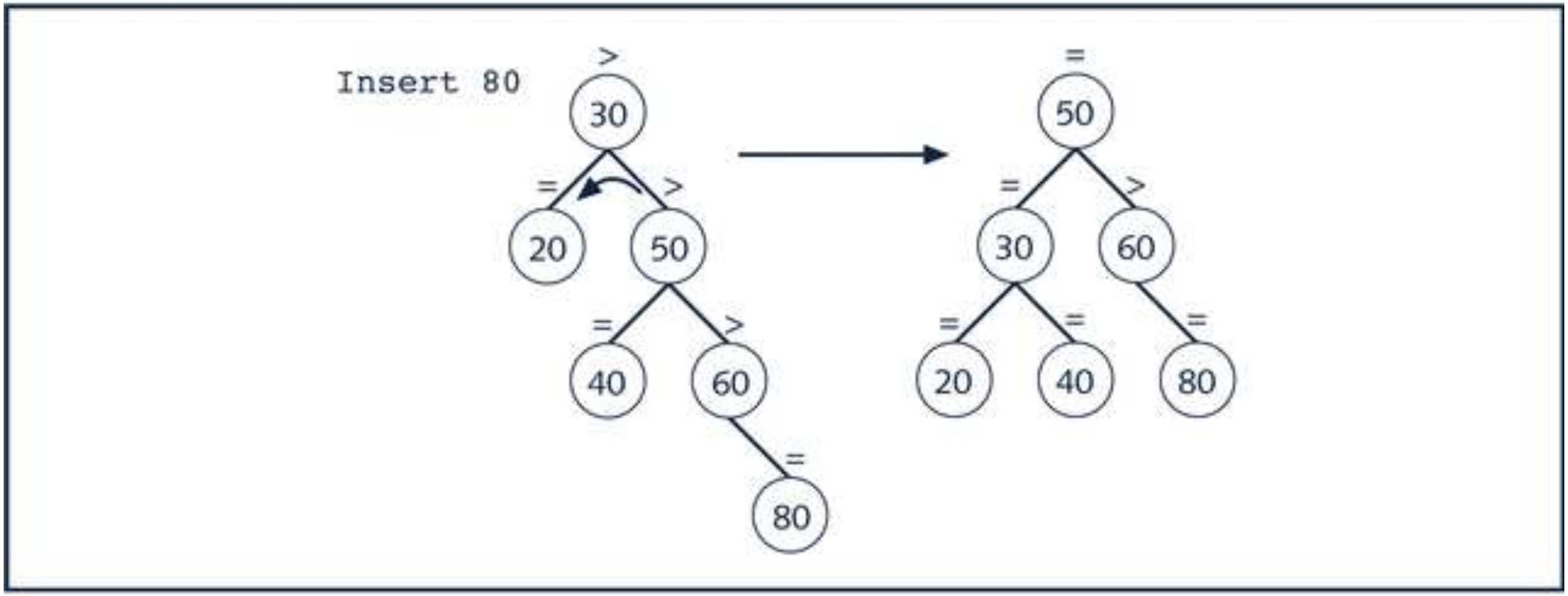


Figure 11-43 AVL tree after inserting 80

AVL Tree Rotations

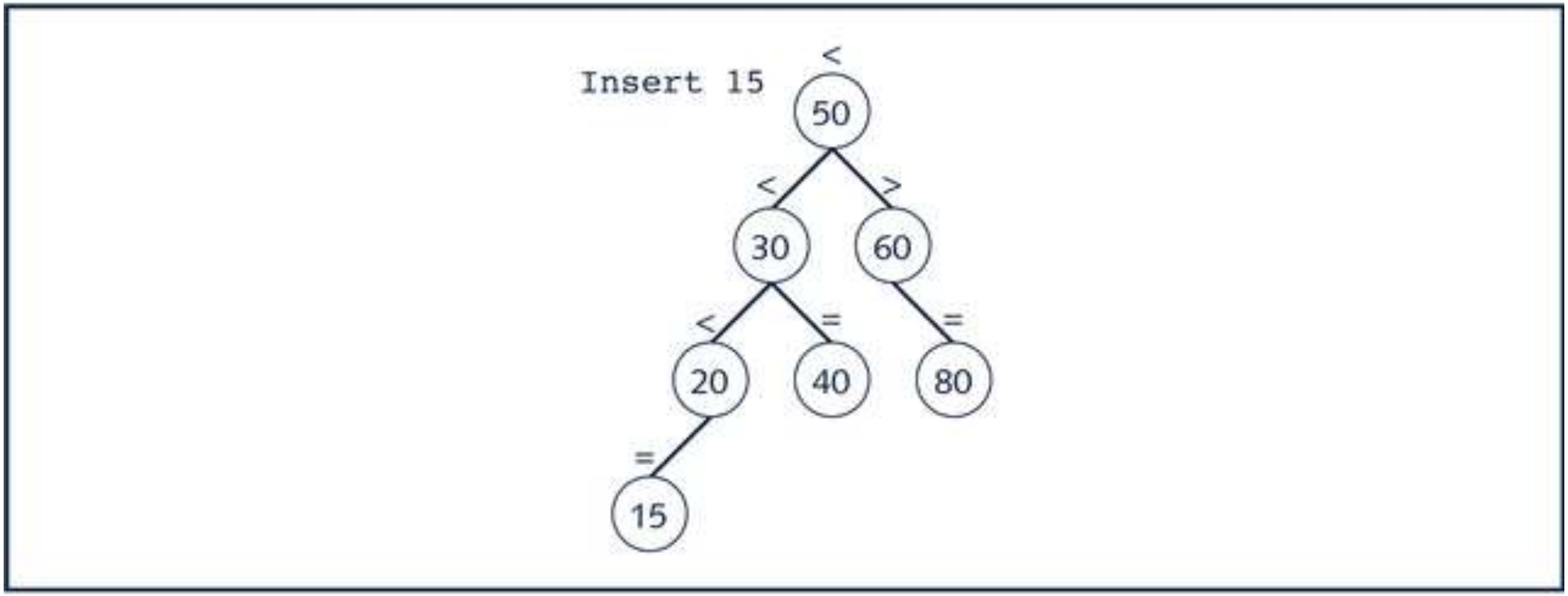


Figure 11-44 AVL tree after inserting 15

AVL Tree Rotations

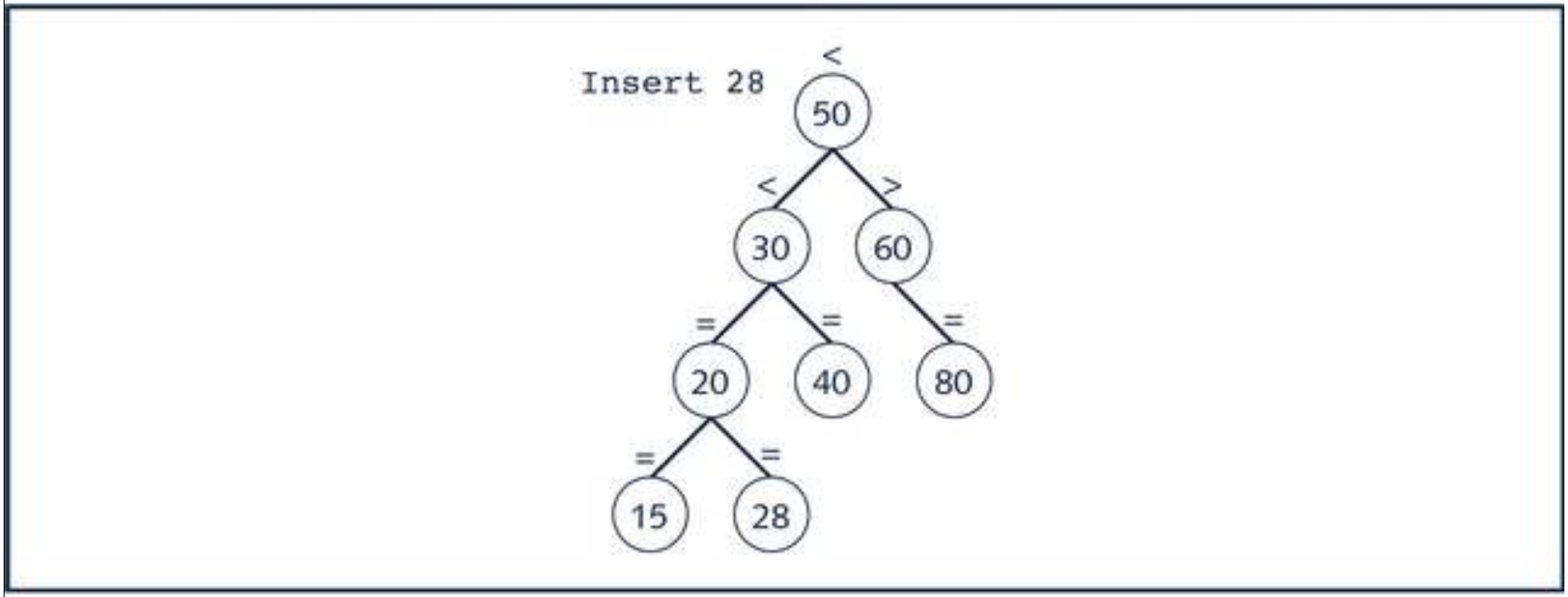


Figure 11-45 AVL tree after inserting 28

AVL Tree Rotations

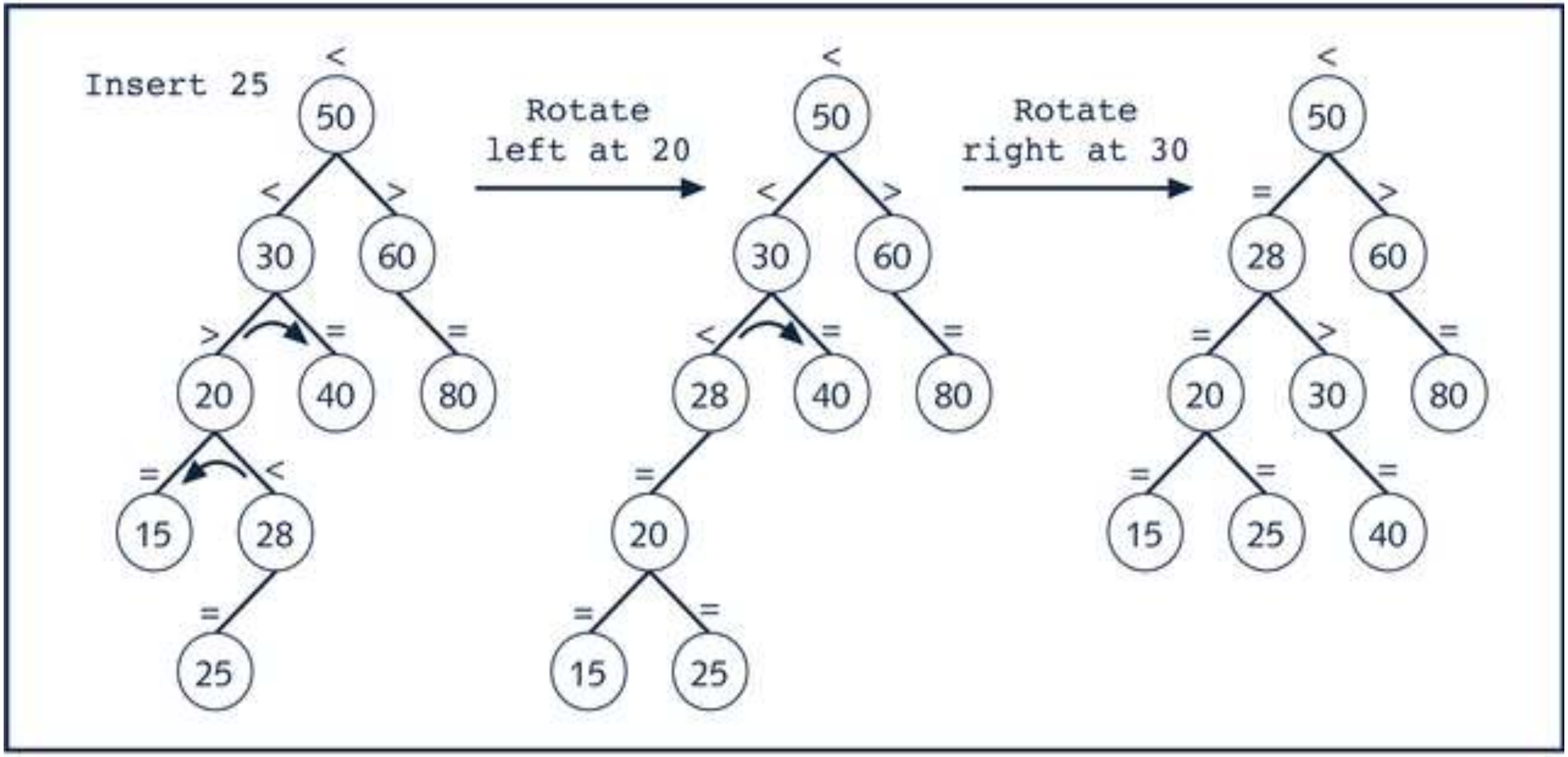


Figure 11-46 AVL tree after inserting 25

Deletion From AVL Trees

- **Case 1:** the node to be deleted is a leaf
- **Case 2:** the node to be deleted has no right child, that is, its right subtree is empty
- **Case 3:** the node to be deleted has no left child, that is, its left subtree is empty
- **Case 4:** the node to be deleted has a left child and a right child

Analysis: AVL Trees

Consider all the possible AVL trees of height h. Let T_h be an AVL tree of height h such that T_h has the fewest number of nodes. Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h. Then:

$$|T_h| = |T_{hl}| + |T_{hr}| + 1$$

where $| T_h |$ denotes the number of nodes in T_h.

Analysis: AVL Trees

Suppose that T_{hl} is of height $h – 1$ and T_{hr} is of height $h – 2$. T_{hl} is an AVL tree of height $h – 1$ such that T_{hl} has the fewest number of nodes among all AVL trees of height $h – 1$. T_{hr} is an AVL tree of height $h – 2$ that has the fewest number of nodes among all AVL trees of height $h – 2$. T_{hl} is of the form T_h -1 and T_{hr} is of the form T_h -2. Hence:

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

$$\begin{aligned} |T_0| &= 1 \\ |T_1| &= 2 \end{aligned}$$

Analysis: AVL Trees

Let *Fh*+2 = |*Th* | + 1. Then:

$$\begin{aligned} F_{h+2} &= F_{h+1} + F_h \\ F_2 &= 2 \\ F_3 &= 3. \end{aligned}$$

Called a Fibonacci sequence; solution to *Fh* is given by:

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \quad \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

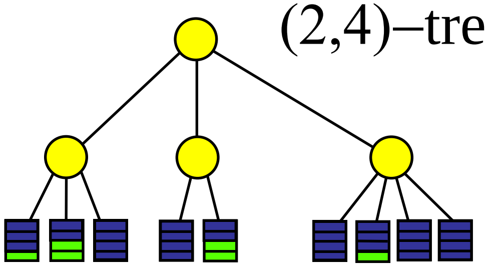
Hence $|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^{h+2}$

From this it can be concluded that $h \approx (1.44)\log_2 |T_h|$

Lecture 9

(a,b)-tree (or B-tree)

- *T* is an (*a*,*b*)-tree (*a*≥2 and *b*≥2*a*-1)
  - All leaves on the same level (contain between *a* and *b* elements)
  - Except for the root, all nodes have degree between *a* and *b*
  - Root has degree between 2 and *b*


(2,4)–tree

nt $O(\log_a N)$

ed in one disk block

$\Theta(B)$

$O(\log_B N)$

$O(N/B)$ space and

query

(*a*,*b*)-Tree Insert

- Insert:

Search and insert element in leaf *v*

DO *v* has *b+1* elements $\left\lceil \frac{b+1}{2} \right\rceil \le b$ $\left\lfloor \frac{b+1}{2} \right\rfloor \ge a$

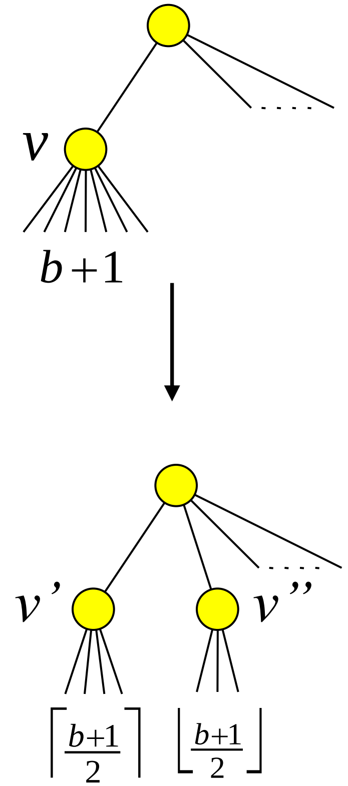
Split *v*:

make nodes *v'* and *v''* with

and $O(\log_a N)$

elements

insert element (ref) in



(*a*,*b*)-Tree Delete

- Delete:

Search and delete element from leaf *v*

DO *v* has *a-1* children

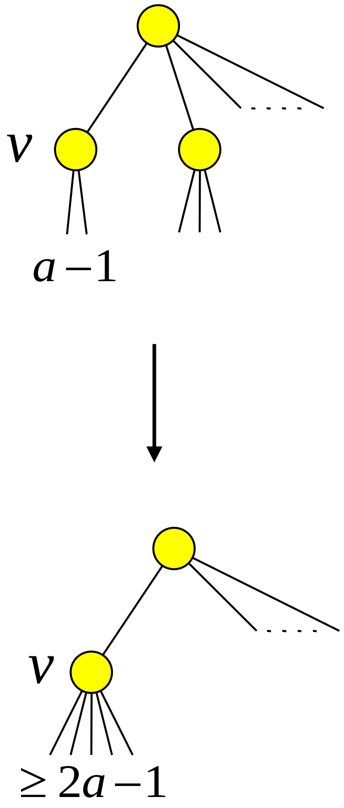
Fuse *v* with sibling *v'*:

move children of *v'* to *v*

delete element (ref) from *parent(v)*

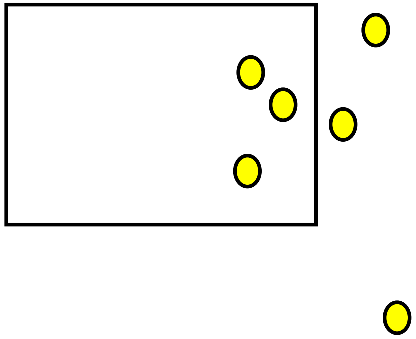
(delete root if necessary)

$O(\log_a N)$



Range Searching in 2D

- Recall the definition: given a set of n points, build a data structure that for any query rectangle R, reports all points in R
- Updates are also possible, but:
  - Fairly complex in theory
  - Straightforward approach



Lecture 10-11

Graph

Königsberg Bridge Problem

In 1736, the following problem was posed:

- River Pregel (Pregolya) flows around the island Kneiphof
- Divides into two
- River has four land areas (*A*, *B*,*C*, D)
- Bridges are labeled *a*, *b*, *c*, *d*, *e*, *f*, *g*

Graphs

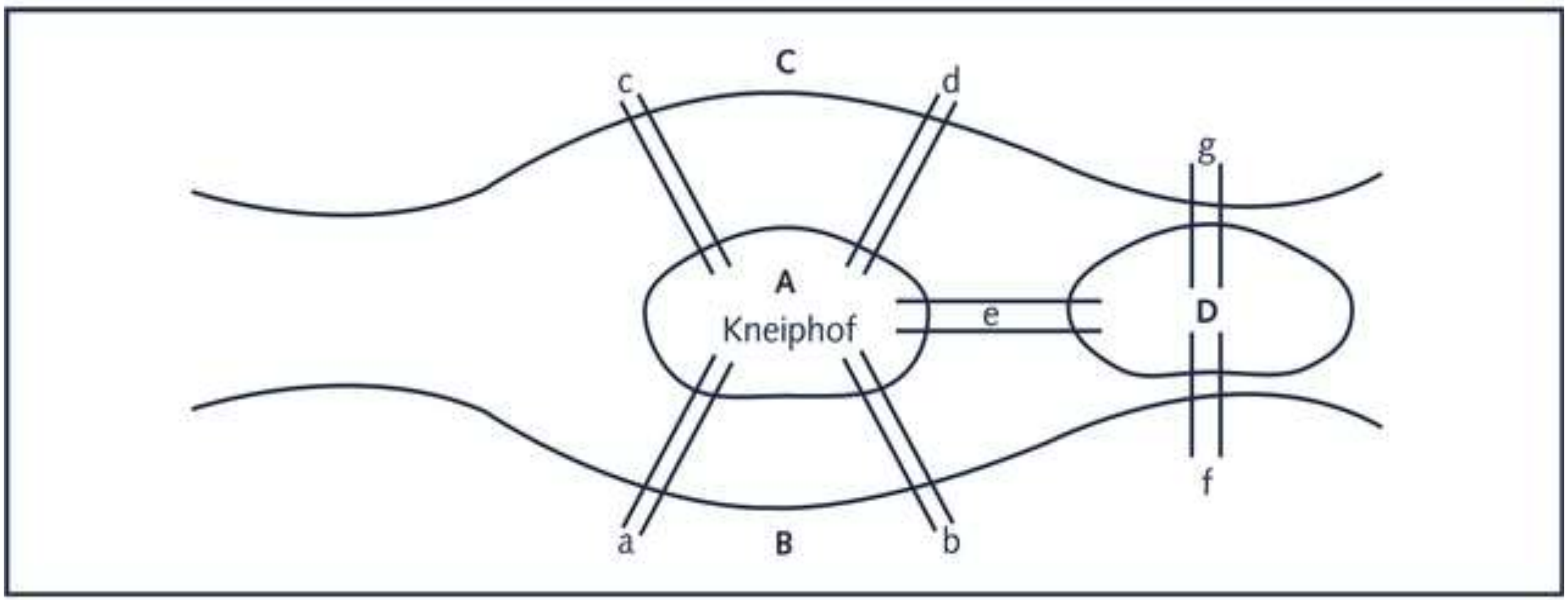


Figure 12-1 Königsberg bridge problem

Königsberg Bridge Problem

- The Königsberg bridge problem
    - Starting at one land area, is it possible to walk across all the bridges exactly once and return to the starting land area?
- In 1736, Euler represented Königsberg bridge problem as graph; Answered the question in the negative.
- This marked (as recorded) the birth of graph theory.

Graphs

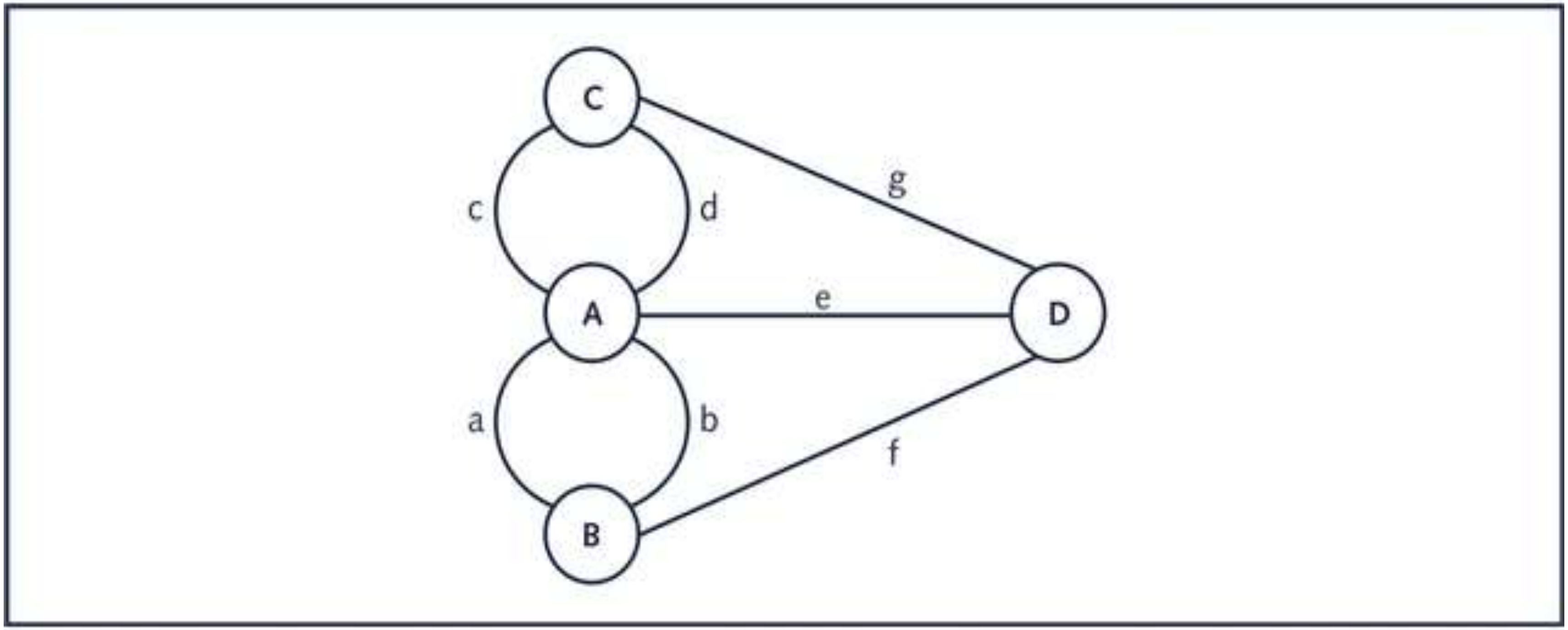


Figure 12-2 Graph representation of the Königsberg bridge problem

Graph Definitions and Notation

- A **graph** *G* is a pair,*g* = (*V*, *E*), where *V* is a finite nonempty set, called the set of **vertices** of *G*, and *E* $\subseteq$ *V x V*
- Elements of *E* are the pair of elements of *V*. *E* is called the set of **edges**

Graph Definitions and Notation

- Let *V*(*G*) denote the set of vertices, and *E*(*G*) denote the set of edges of a graph *G*. If the elements of *E*(*G*) are ordered pairs, *g* is called a **directed graph** or **digraph**; Otherwise, *g* is called an **undirected graph**
- In an undirected graph, the pairs (*u*, *v*) and (*v*, *u*) represent the same edge

Various Undirected Graphs

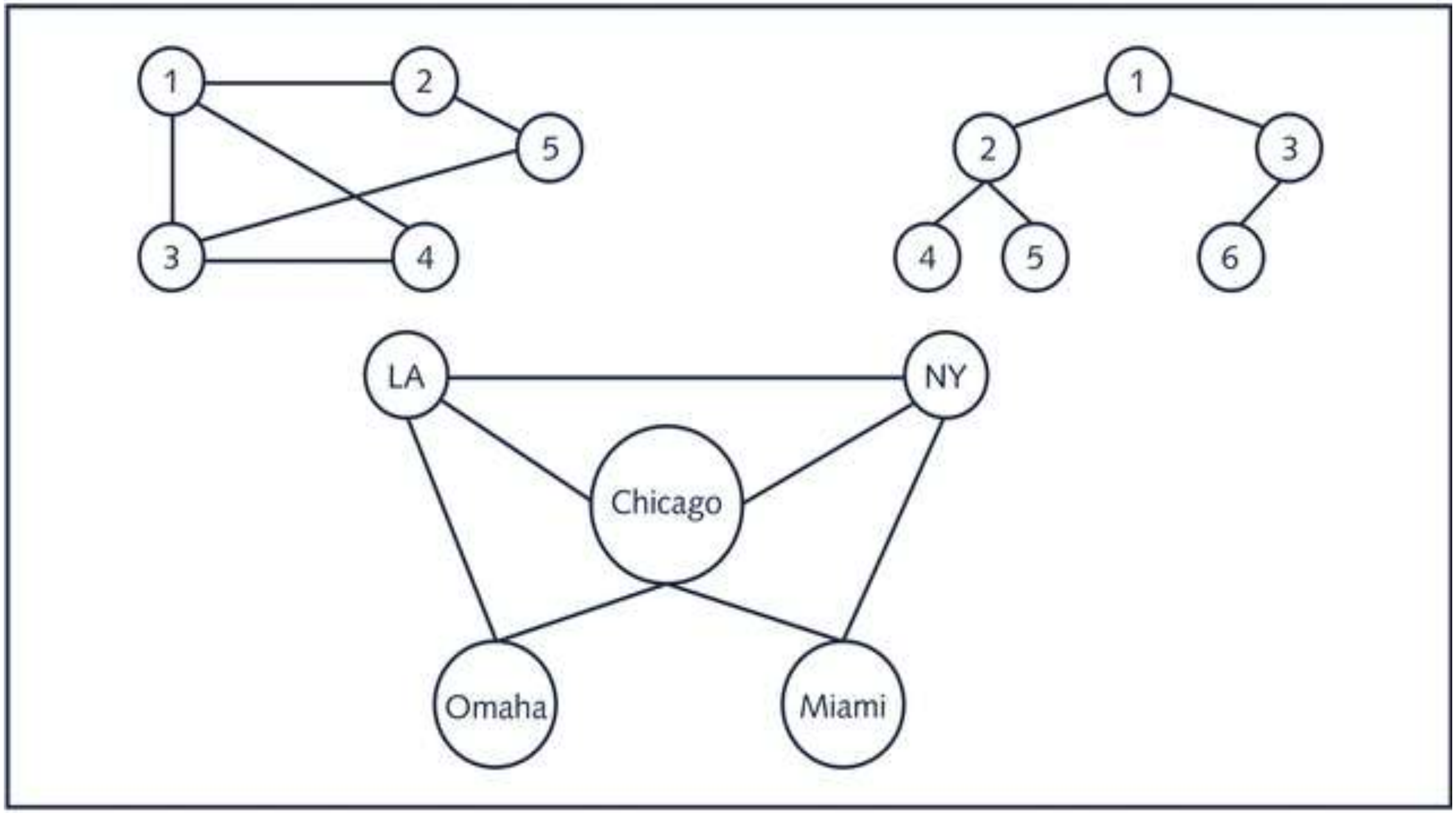


Figure 12-3 Various undirected graphs

Various Directed Graphs

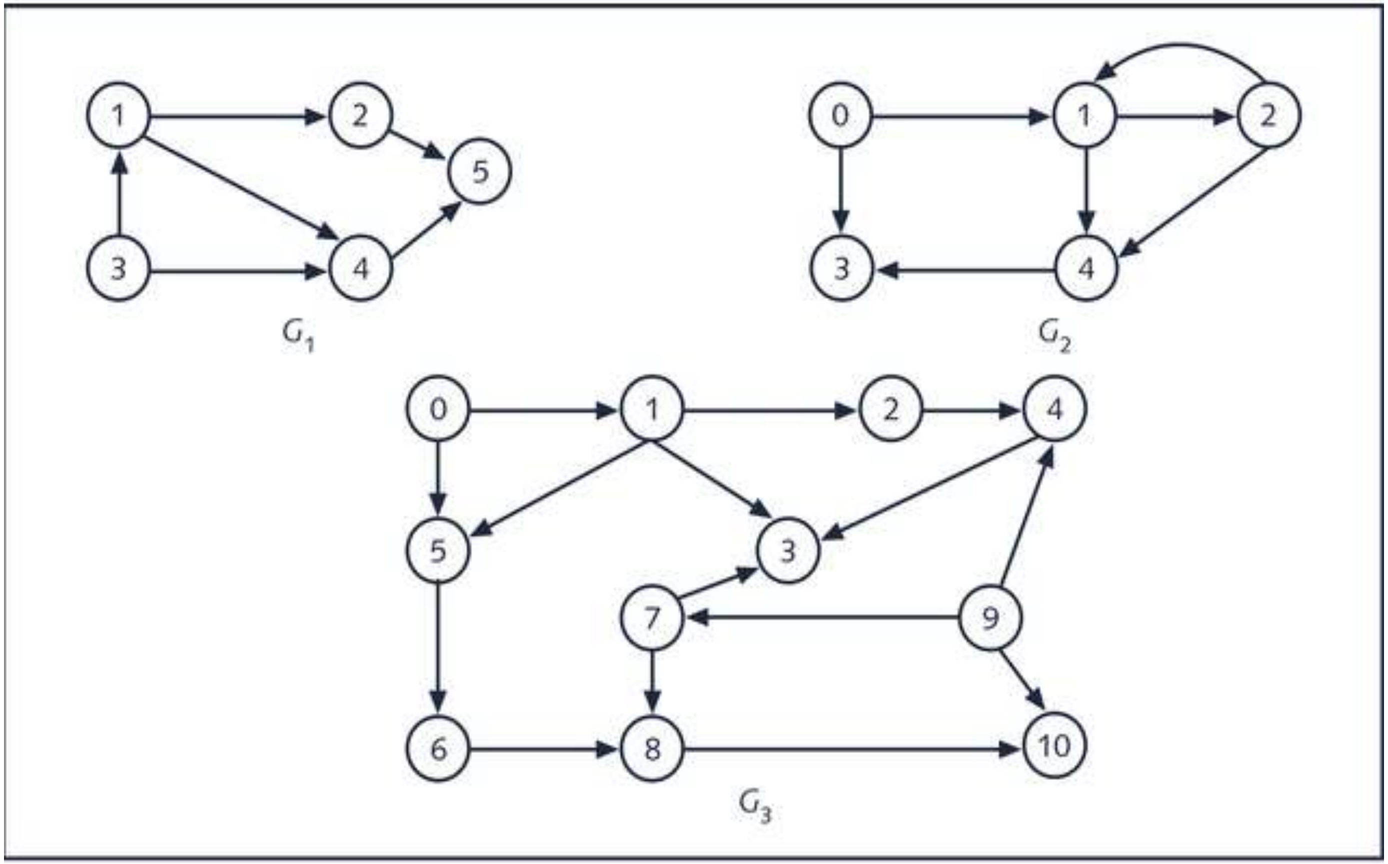


Figure 12-4 Various directed graphs

Graph Representation: Adjacency Matrix

- Let *G* be a graph with *n* vertices, where $n > 0$
- Let $V(G) = \{v_1, v_2, ..., v_n\}$
- The adjacency matrix *AG* is a two-dimensional $n \times n$ matrix such that the (*i*, *j*)th entry of *AG* is 1 if there is an edge from *vi* to *vj*; otherwise, the (*i*, *j*)th entry is zero

Graph Representation: Adjacency Matrix

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A_{G_3} = \begin{array}{c|ccccccccccc|} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 9 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Graph Representation: Adjacency Lists

- In adjacency list representation, corresponding to each vertex, *v*, is a linked list such that each node of the linked list contains the vertex *u*, such that $(v, u) \in E(G)$
- Array, *A*, of size *n*, such that *A*[*i*] is a pointer to the linked list containing the vertices to which *vi* is adjacent
- Each node has two components, (vertex and link)
- Component vertex contains index of vertex adjacent to vertex *i*

Graph Representation: Adjacency Matrix

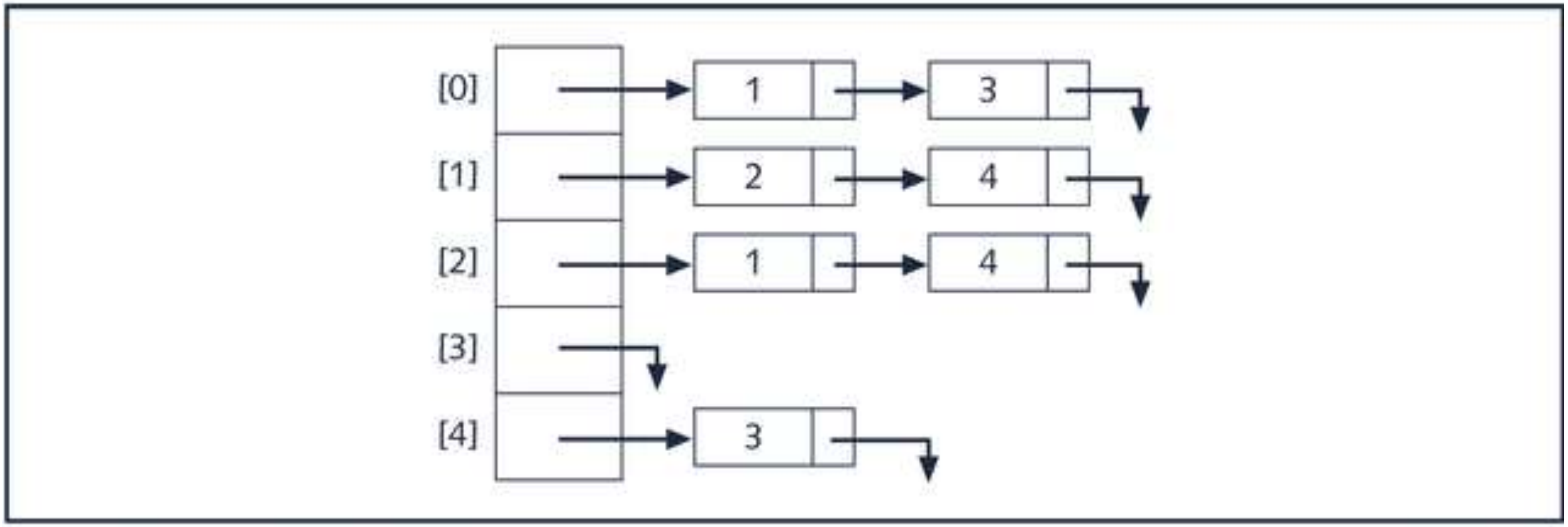


Figure 12-5 Adjacency list of graph G_2 of Figure 12-4

Graph Representation: Adjacency Matrix

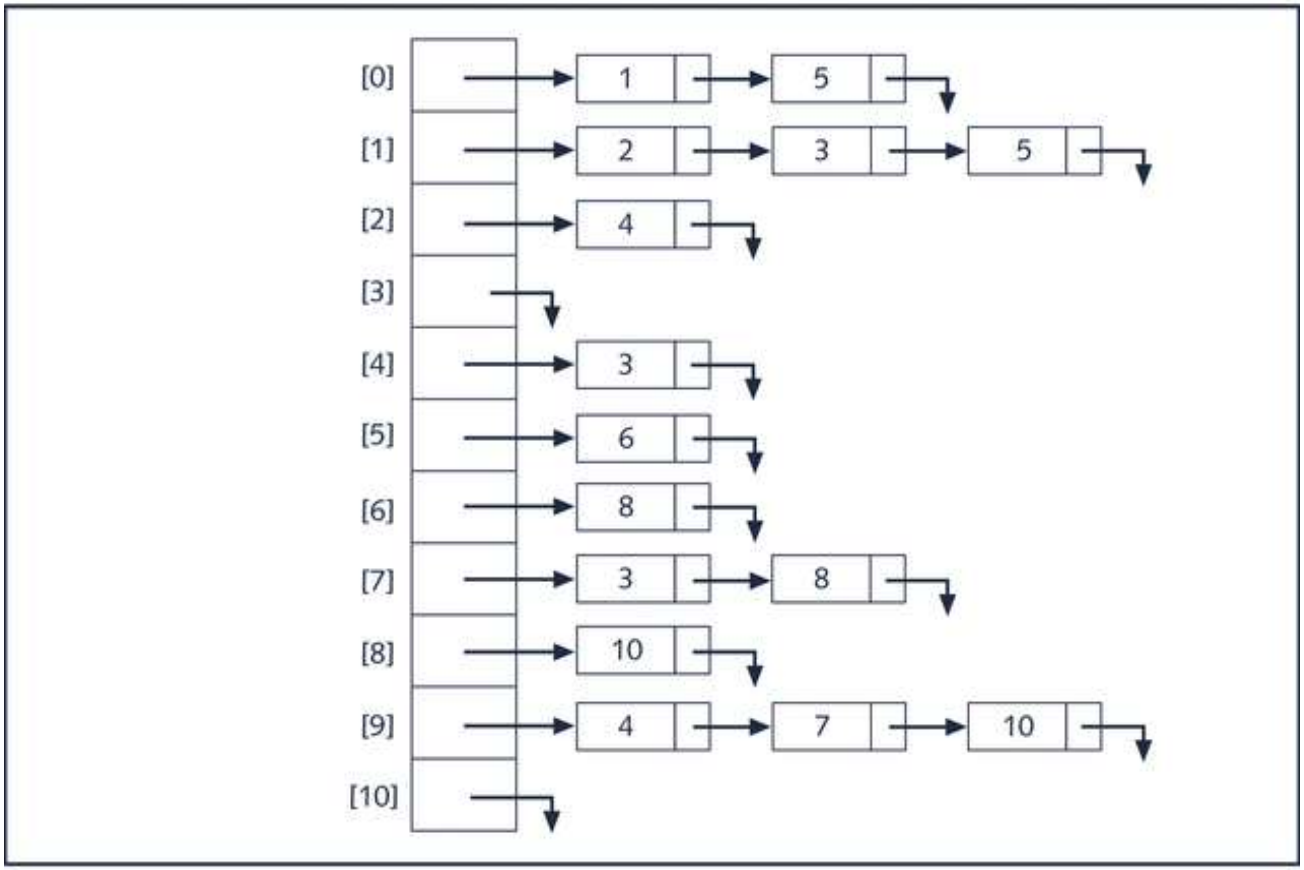


Figure 12-6 Adjacency list of graph G_3 of Figure 12-4

Operations on Graphs

- Create the graph: store in memory using a particular graph representation
- Clear the graph: make the graph empty
- Determine whether the graph is empty
- Traverse the graph
- Print the graph

class linkedListGraph

```
template<class vType>
class linkedListGraph: public linkedListType<vType>
{
public:
   void getAdjacentVertices(vType adjacencyList[],
                                int& length);
     //Function to retrieve the vertices adjacent to a given
     //vertex.
     //Postcondition: The vertices adjacent to a given vertex
     //               are retrieved in the array
                      adjacencyList. The parameter length
                      specifies the number
     //               of vertices adjacent to a given vertex.
};
```

class linkedListGraph

```
template<class vType>
void linkedListGraph<vType>::getAdjacentVertices
                         (vType adjacencyList[], int& length)
{
   nodeType<vType> *current;
   length = 0;
   current = first;

   while(current != NULL)
   {
        adjacencyList[length++] = current->info;
        current = current->link;
   }
}
```

Templates

```
template<class elemType, int size>
class listType
{
public:
   .
   .
   .
private:
   int maxSize;
   int length;
   elemType listElem[size];
};
```

class Template

- This class template contains an array data member
- Array element type and size of array passed as parameters to class template
- To create a list of 100 components of int elements:

```
listType<int, 100> intList;
```

- Element type and size of array both passed to class template listType

Lecture 12

Graph Traversals

- Depth first traversal
  - – Mark node v as visited
  - – Visit the node
  - – For each vertex u adjacent to v
    - If u is not visited
      - – Start the depth first traversal at u

Depth First Traversal

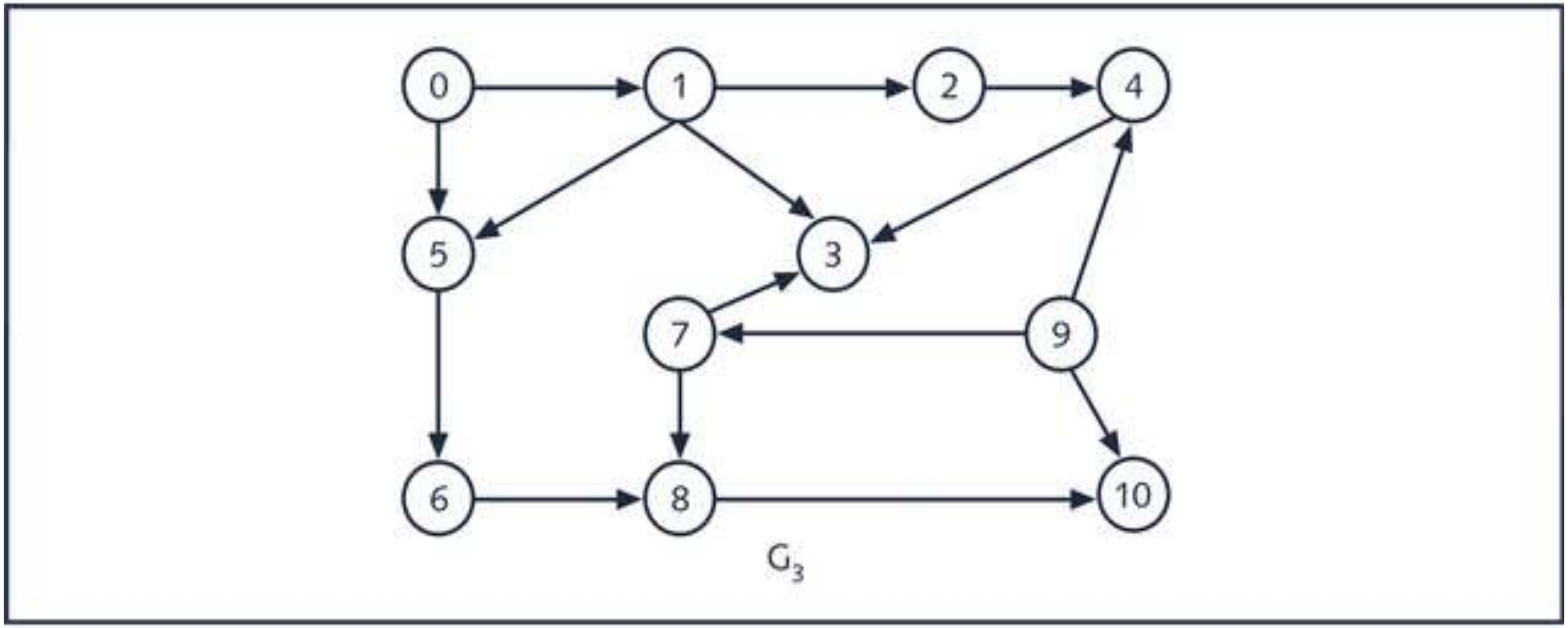


Figure 12-7 Directed graph G_3

Breadth First Traversal

```
The general algorithm is:
   a. for each vertex v in the graph
        if v is not visited
               add v to the queue //start the breadth
                                  // first search at v
   b. Mark v as visited
   c. while the queue is not empty
        c.1. Remove vertex u from the queue
        c.2. Retrieve the vertices adjacent to u
        c.3. for each vertex w that is adjacent to u
               if w is not visited
               c.3.1. Add w to the queue
               c.3.2. Mark w as visited
```

Graph Traversals

Graph Traversals

0 1 2 4 3 5 6 8 10 7 9
0 1 5 2 3 6 4 8 10 7 9

Lecture 13

Shortest Path Algorithm

- **Weight of the edge:** edges connecting two vertices can be assigned a nonnegative real number
- **Weight of the path** *P:* sum of the weights of all the edges on the path *P*; **Weight** of *v* from *u* via *P*
- **Shortest path:** path with smallest weight
- **Shortest path algorithm:** greedy algorithm developed by Dijkstra

Shortest Path Algorithm

Let *G* be a graph with *n* vertices, where *n* > 0.

Let *V*(*G*) = {*v*1, *v*2, ..., *vn*}. Let *W* be a two-dimensional *n X n* matrix such that:

$$W(i,j) = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \text{ is an edge in } G \text{ and } w_{ij} \text{ is the weight of the edge } (v_i, v_j) \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

Shortest Path

The general algorithm is:

```
1.  Initialize the array smallestWeight so that
      smallestWeight[u] = weights[vertex, u]
2.  Set smallestWeight[vertex] = 0
3.  Find the vertex, v, that is closest to vertex for which the shortest path has not been determined
4.  Mark v as the (next) vertex for which the smallest weight is found
5.  For each vertex w in G, such that the shortest path from vertex to w has not been determined and an edge (v, w) exists, if the weight of the path to w via v is smaller than its current weight, update the weight of w to the weight of v + the weight of the edge (v, w)
```

Because there are n vertices, repeat steps 3 through 5 *n* – 1 times

Shortest Path

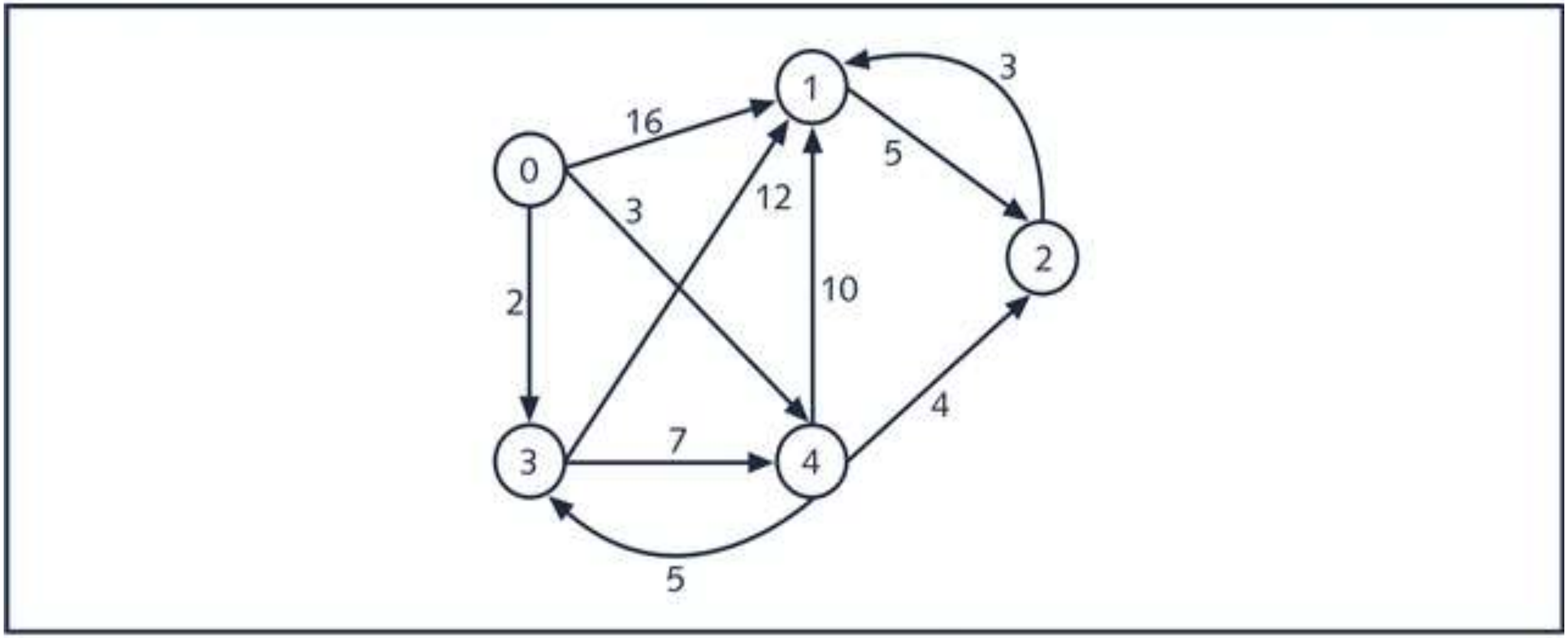
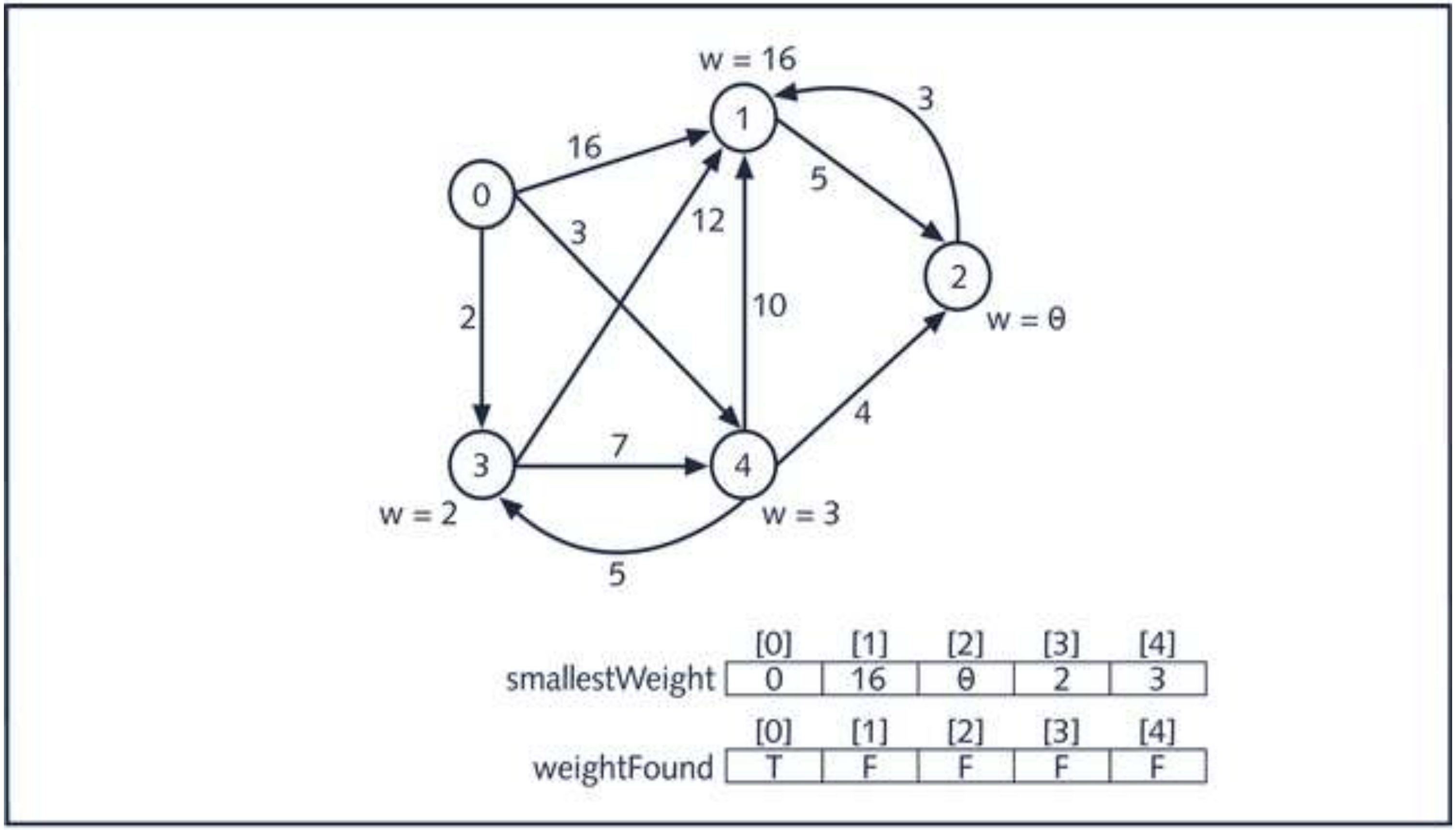


Figure 12-8 Weighted graph *G*

Shortest Path

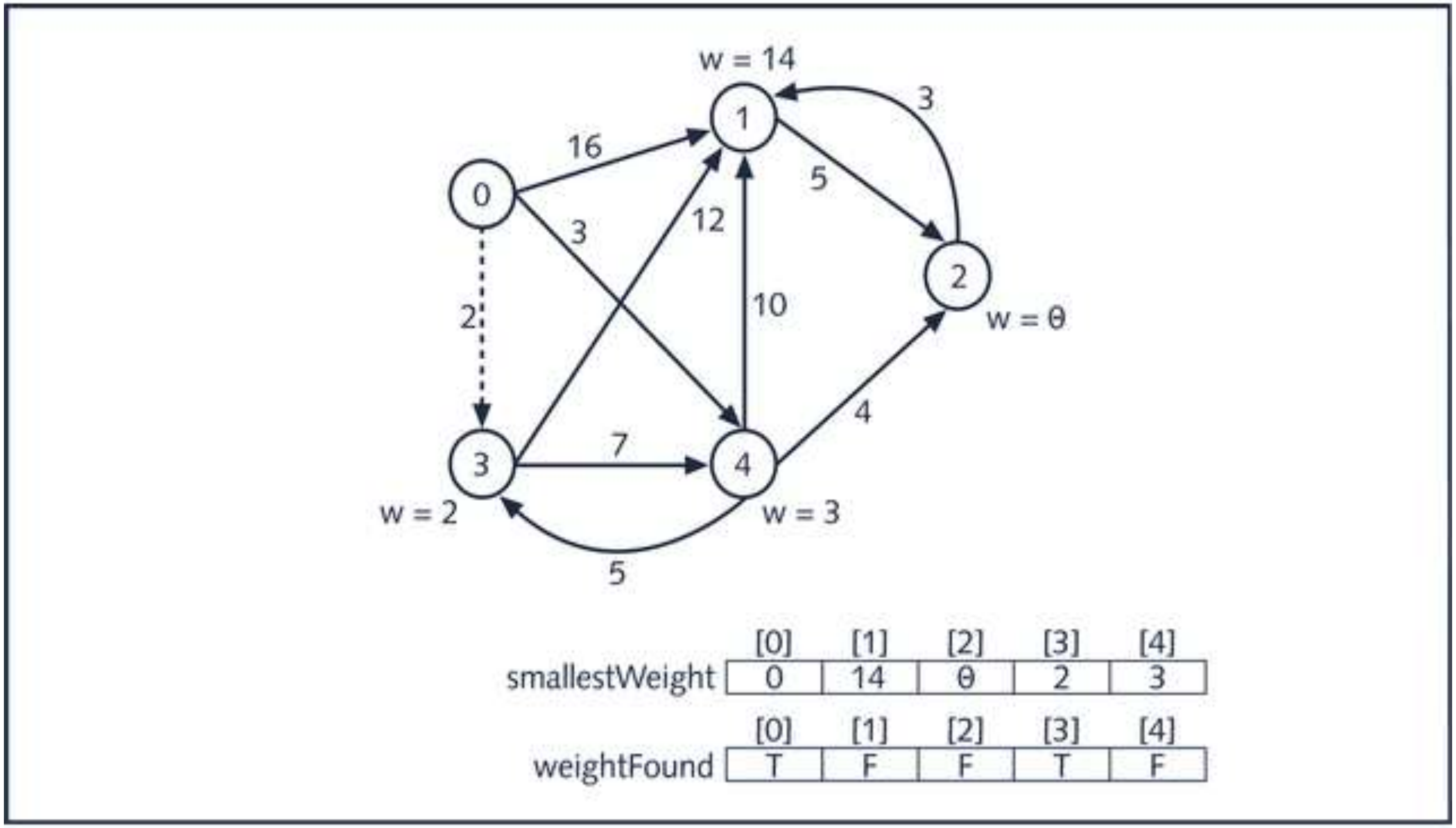


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	16	θ	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	F	F	F

Figure 12-9 Graph after Steps 1 and 2 execute

Shortest Path

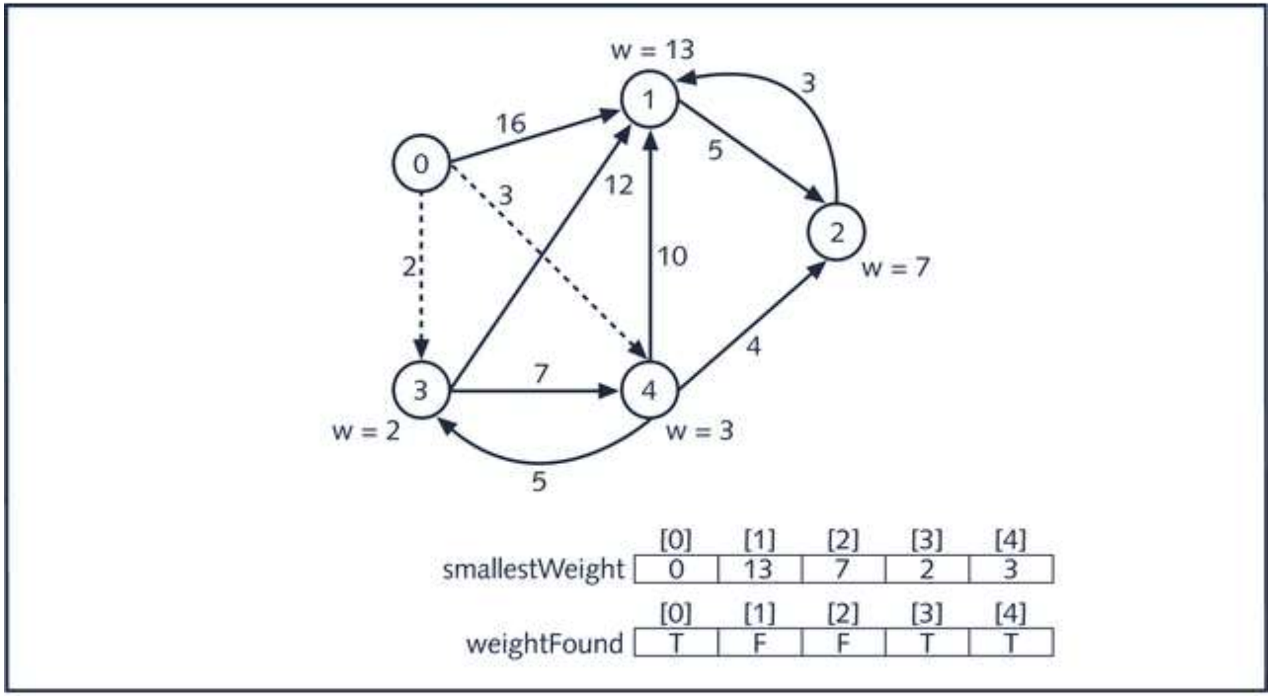


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	14	0	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	F	T	F

Figure 12-10 Graph after the first iteration of Steps 3, 4, and 5

Shortest Path

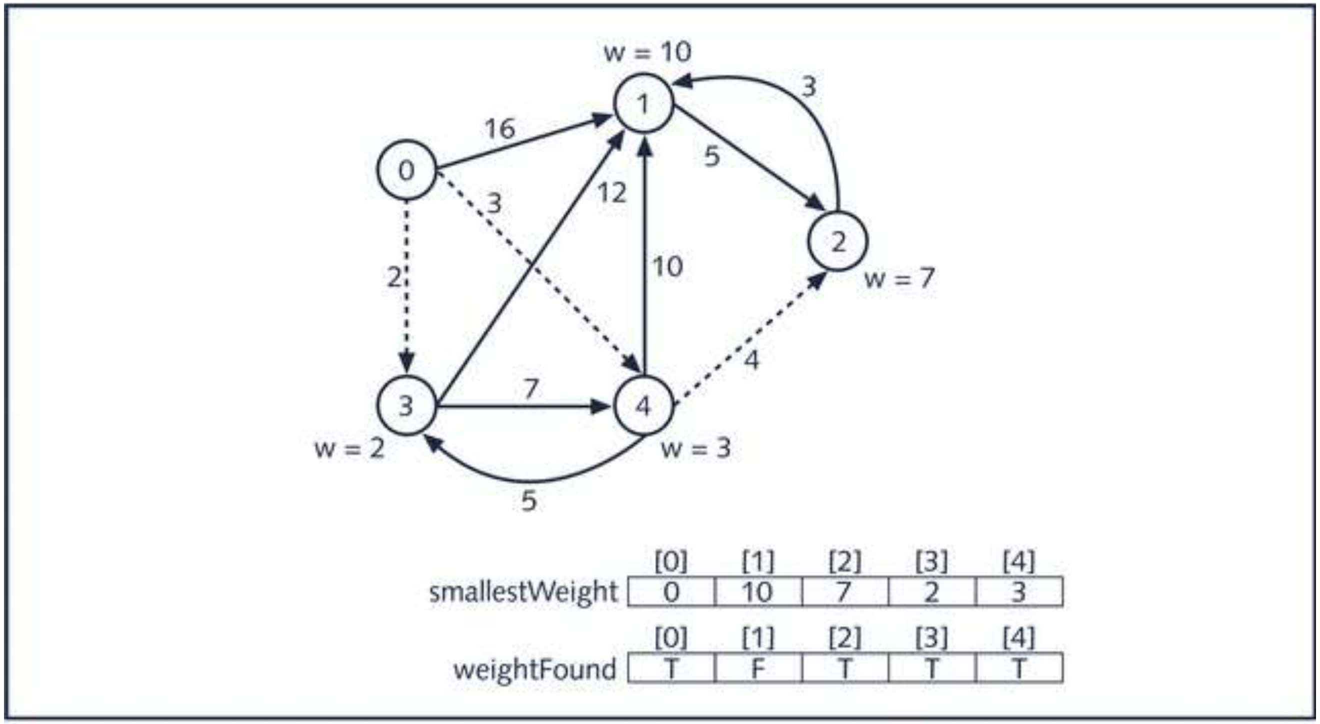


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	13	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	F	T	T

Figure 12-11 Graph after the second iteration of Steps 3, 4, and 5

Shortest Path

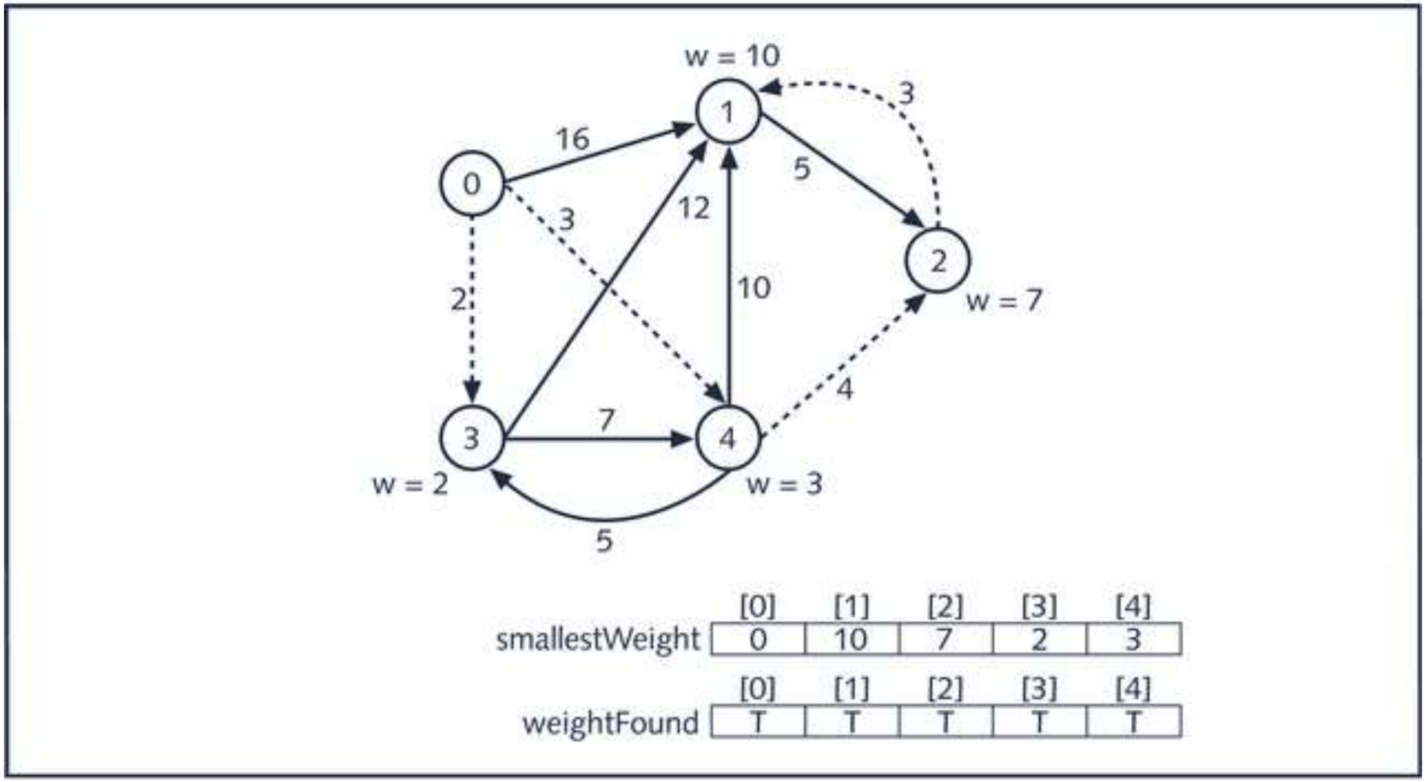


	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	F	T	T	T

Figure 12-12 Graph after the third iteration of Steps 3, 4, and 5

Shortest Path



	[0]	[1]	[2]	[3]	[4]
smallestWeight	0	10	7	2	3

	[0]	[1]	[2]	[3]	[4]
weightFound	T	T	T	T	T

Figure 12-13 Graph after the fourth iteration of Steps 3, 4, and 5

Applet

Lecture 14

Minimal Spanning Tree

This graph represents the airline connections of a company between seven cities (cost factor shown)

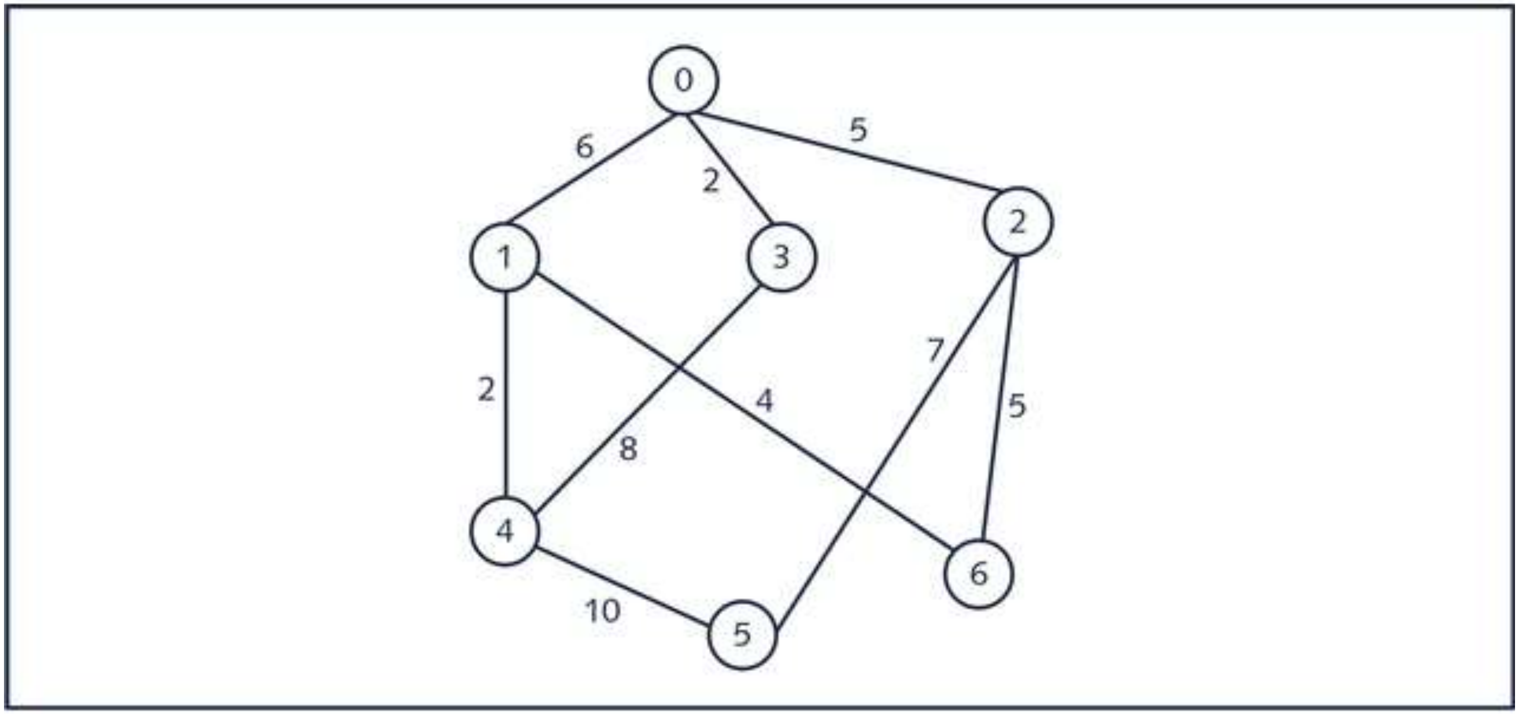


Figure 12-14 Airline connections between cities and the cost factor of maintaining the connections

Minimal Spanning Tree

Company needs to shut down the maximum number of connections and still be able to fly from one city to another (may not be directly).

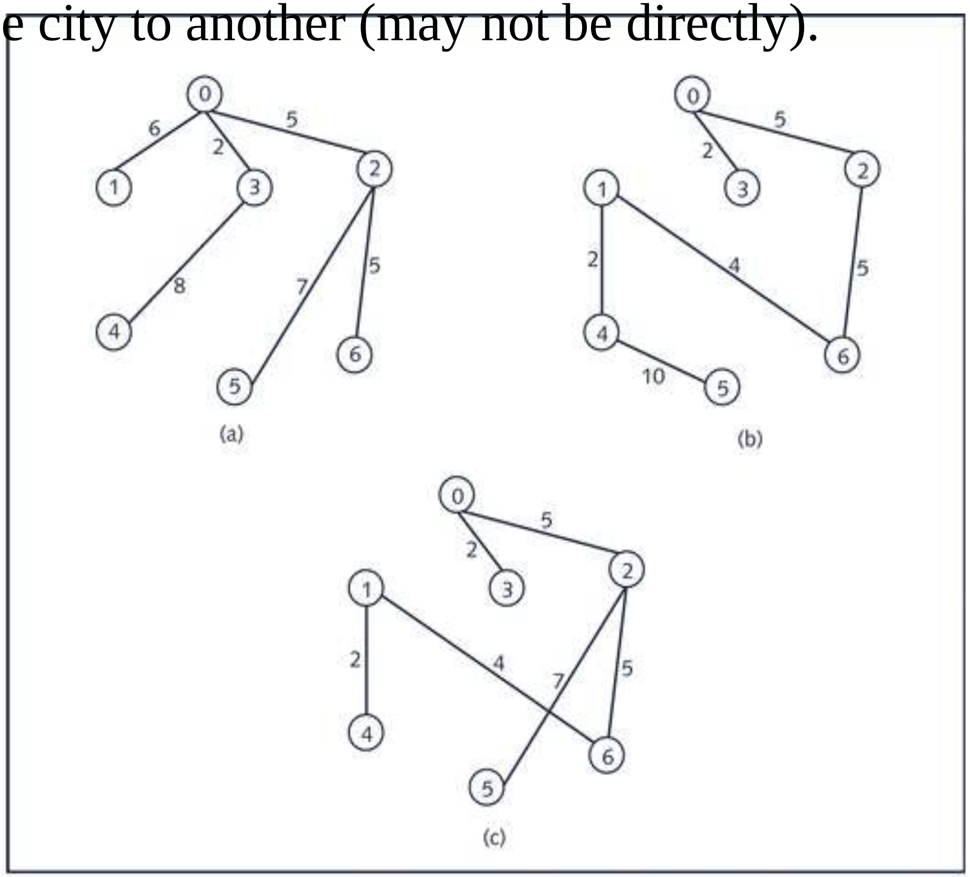


Figure 12-15 Possible solutions to the graph of Figure 12-14

Minimal Spanning Tree

- **(Free) tree** *T :* simple graph such that if *u* and *v* are two vertices in *T*, then there is a unique path from *u* to *v*
- **Rooted tree:** tree in which a particular vertex is designated as a root
- **Weighted tree**: tree in which weight is assigned to the edges in *T*
- If *T* is a weighted tree, the **weight** of *T*, denoted by *W*(*T*), is the sum of the weights of all the edges in *T*

Minimal Spanning Tree

- A tree *T* is called a **spanning tree** of graph *G* if *T* is a subgraph of *G* such that *V*(*T*) = *V*(*G*),
- All the vertices of *G* are in *T*.

Minimal Spanning Tree

- **Theorem:** A graph *G* has a spanning tree if and only if *G* is connected.
- In order to determine a spanning tree of a graph, the graph must be connected.
- Let *G* be a weighted graph. A **minimal spanning tree** of *G* is a spanning tree with the minimum weight.

Prim's Algorithm

- Builds tree iteratively by adding edges until minimal spanning tree obtained
- Start with a source vertex
- At each iteration, new edge that does not complete a cycle is added to tree

Prim’s Algorithm

General form of Prim’s algorithm (let *n* = number of vertices in *G*):

```
1. Set V(T) = {source}
2. Set E(T) = empty
3. for i = 1 to n
    3.1 minWeight = infinity;
    3.2 for j = 1 to n
         if vj is in V(T)
           for k = 1 to n
             if vk is not in T and weight[vj][vk] < minWeight
             {
               endVertex = vk;
               edge = (vj, vk);
               minWeight = weight[vj][vk];
             }
      3.3 V(T) = V(T) ∪ {endVertex};
      3.4 E(T) = E(T) ∪ {edge};
```

Prim’s Algorithm

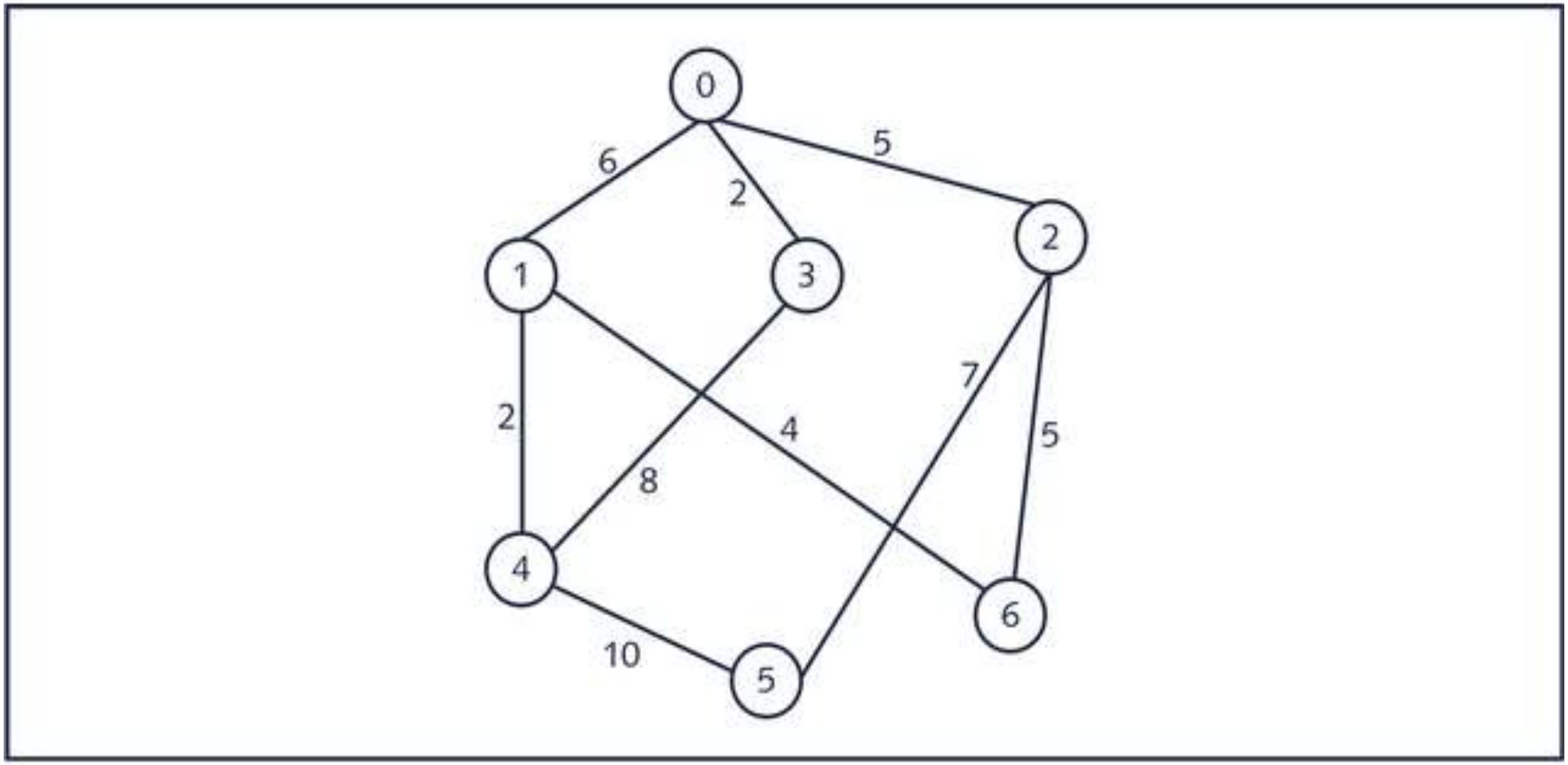
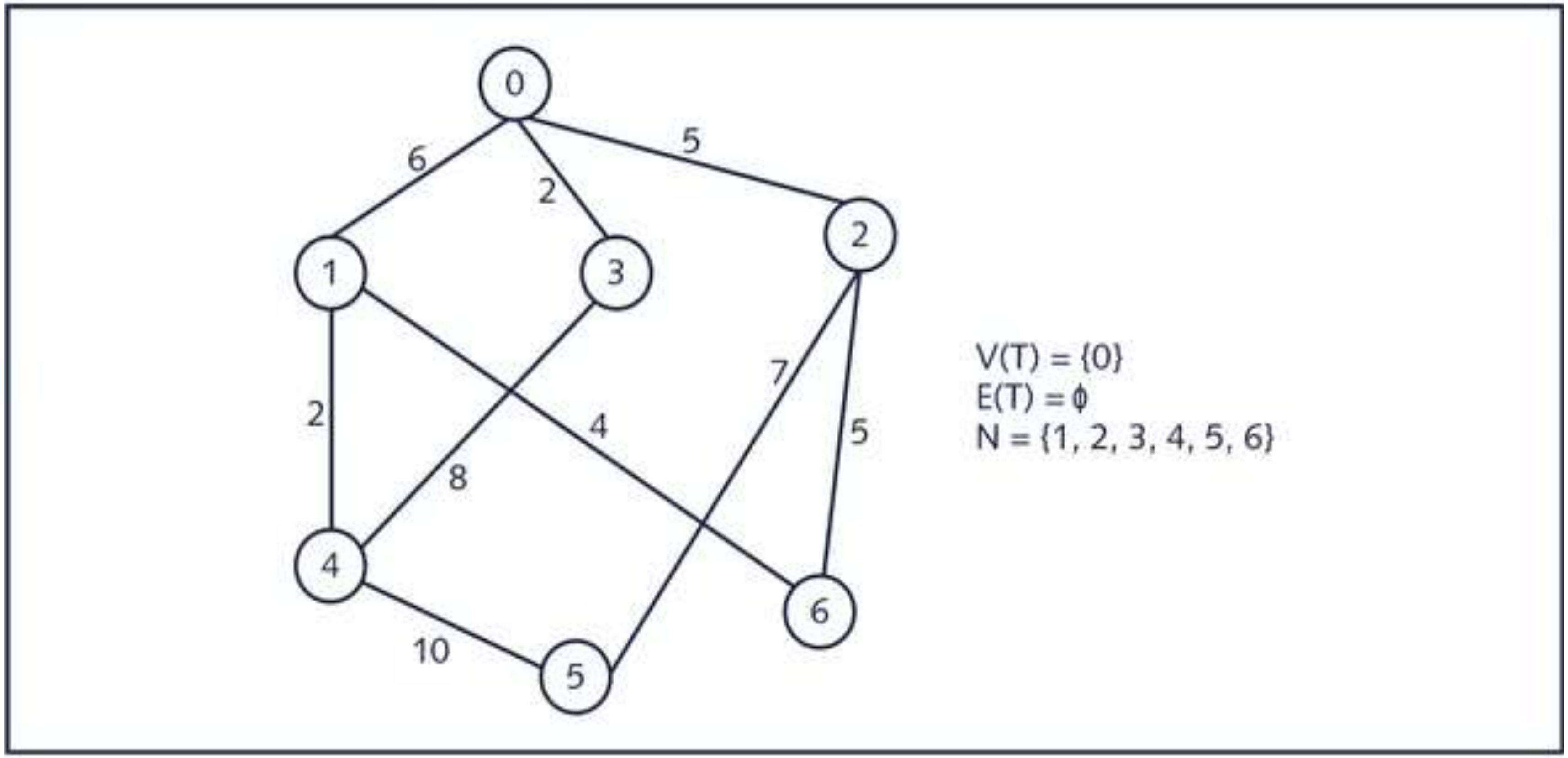


Figure 12-16 Weighted graph *G*

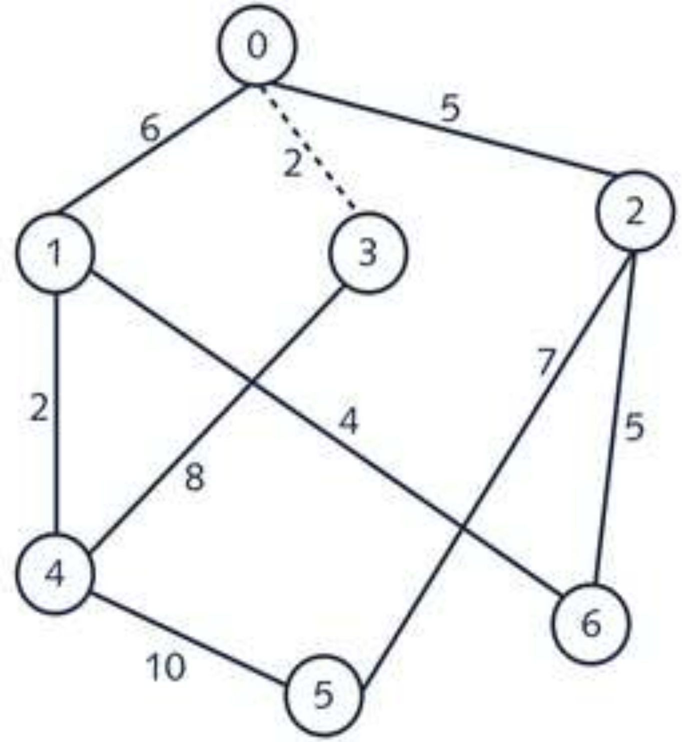
Prim’s Algorithm



$V(T) = \{0\}$
$E(T) = \phi$
$N = \{1, 2, 3, 4, 5, 6\}$

Figure 12-17 Graph *G*, *V(T)*, *E(T)*, and *N* after Steps 1 and 2 execute

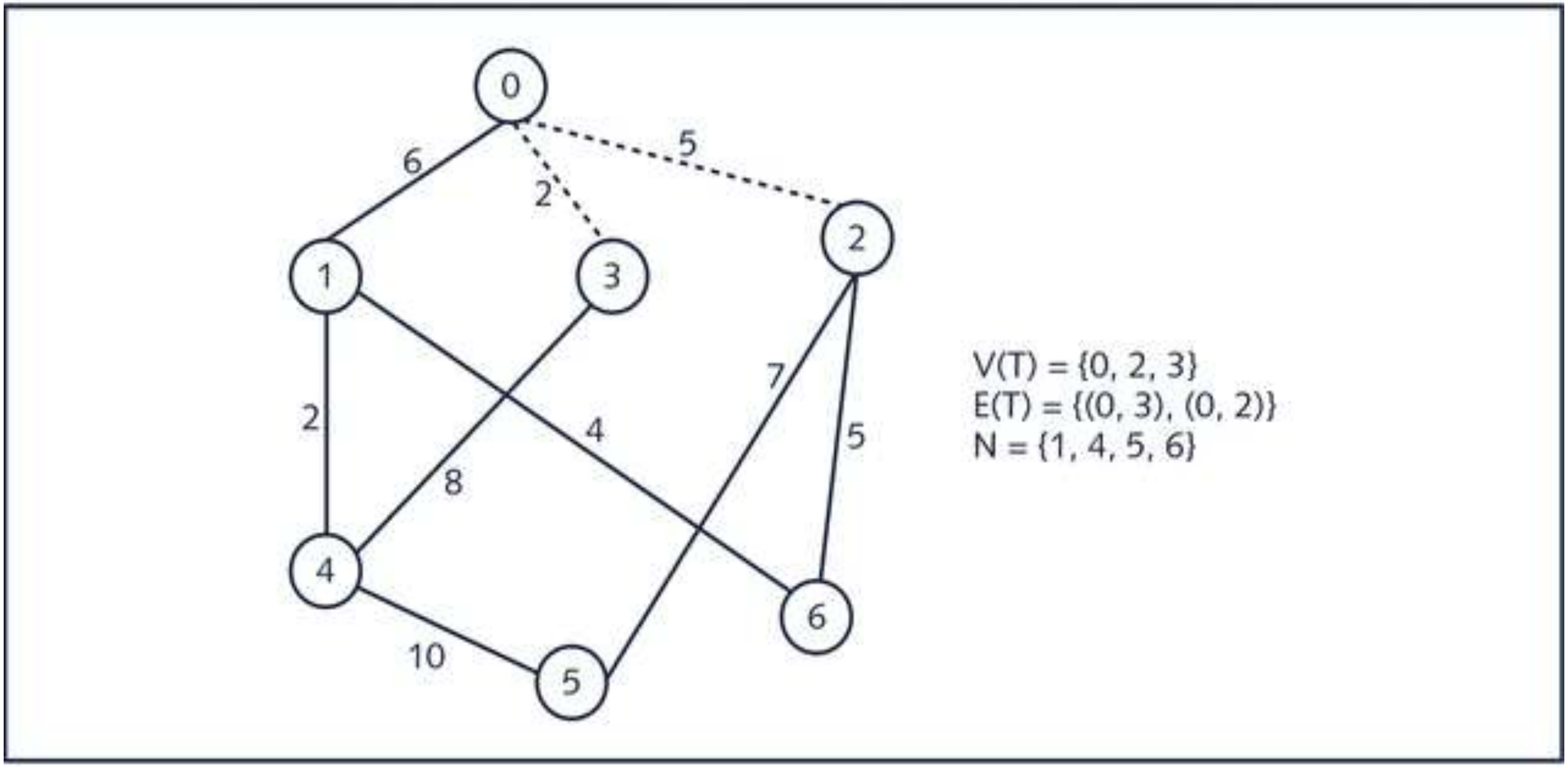
Prim’s Algorithm



V(T) = {0, 3}
E(T) = {(0, 3)}
N = {1, 2, 4, 5, 6}

Figure 12-18 Graph *G*, *V(T)*, *E(T)*, and *N* after the first iteration of Step 3

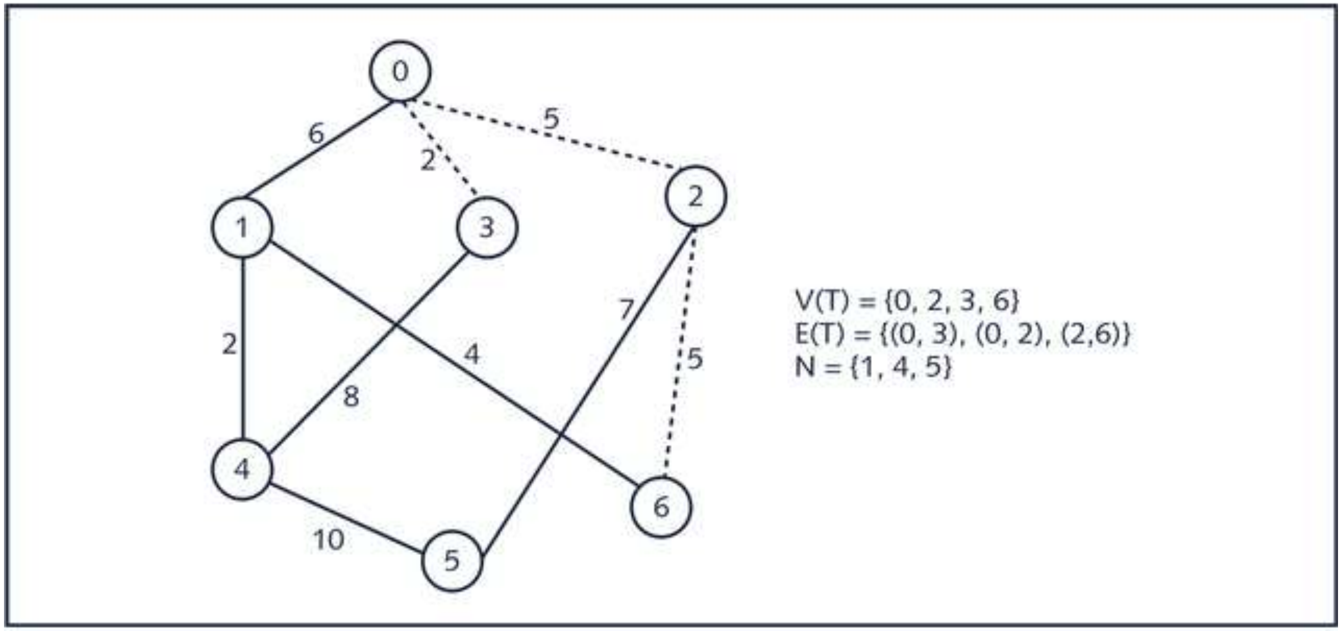
Prim’s Algorithm



V(T) = {0, 2, 3}
E(T) = {(0, 3), (0, 2)}
N = {1, 4, 5, 6}

Figure 12-19 Graph *G*, *V*(*T*), *E*(*T*), and *N* after the second iteration of Step 3

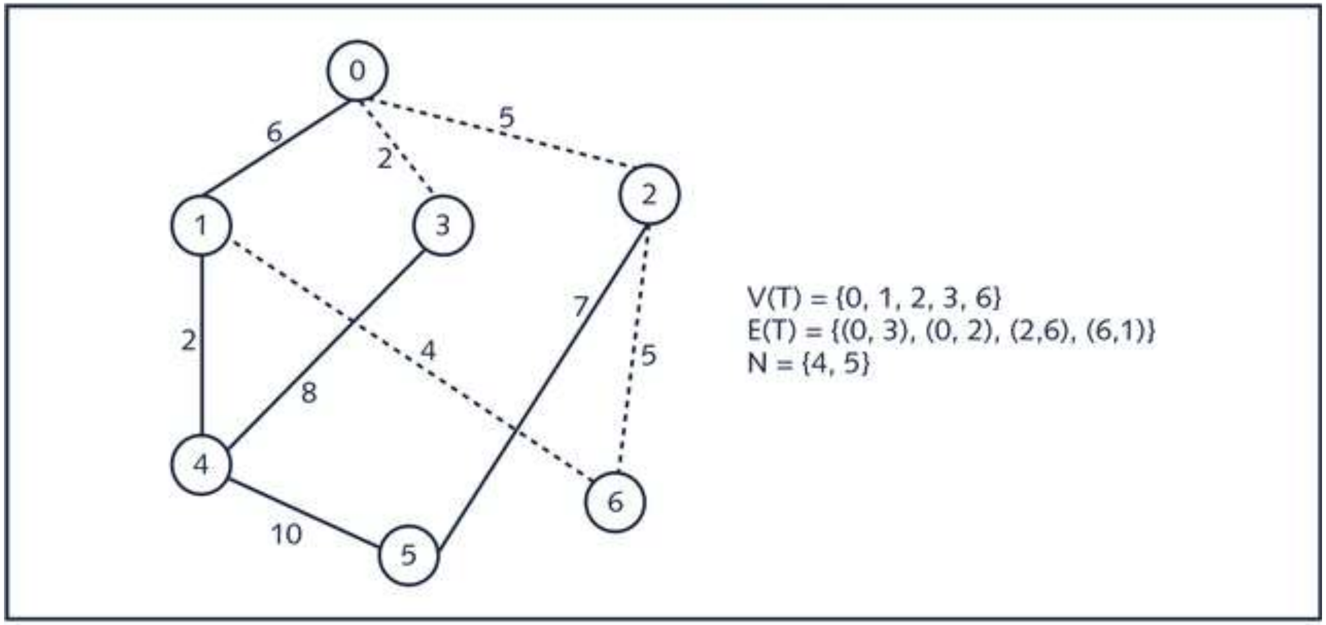
Prim’s Algorithm



V(T) = {0, 2, 3, 6}
E(T) = {(0, 3), (0, 2), (2,6)}
N = {1, 4, 5}

Figure 12-20 Graph *G*, *V*(*T*), *E*(*T*), and *N* after the third iteration of Step 3

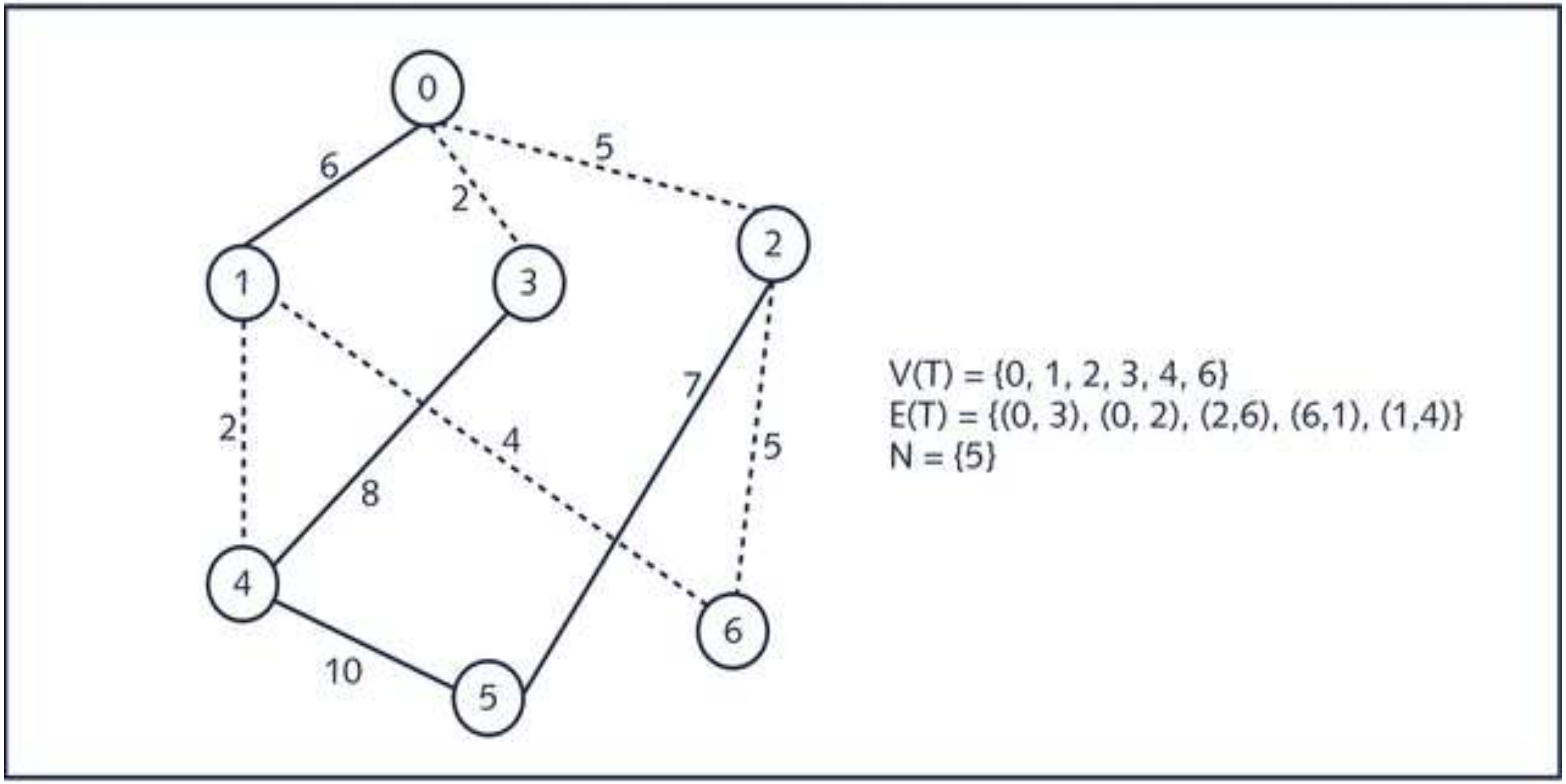
Prim’s Algorithm



V(T) = {0, 1, 2, 3, 6}
E(T) = {(0, 3), (0, 2), (2,6), (6,1)}
N = {4, 5}

Figure 12-21 Graph *G*, *V*(*T*), *E*(*T*), and *N* after the fourth iteration of Step 3

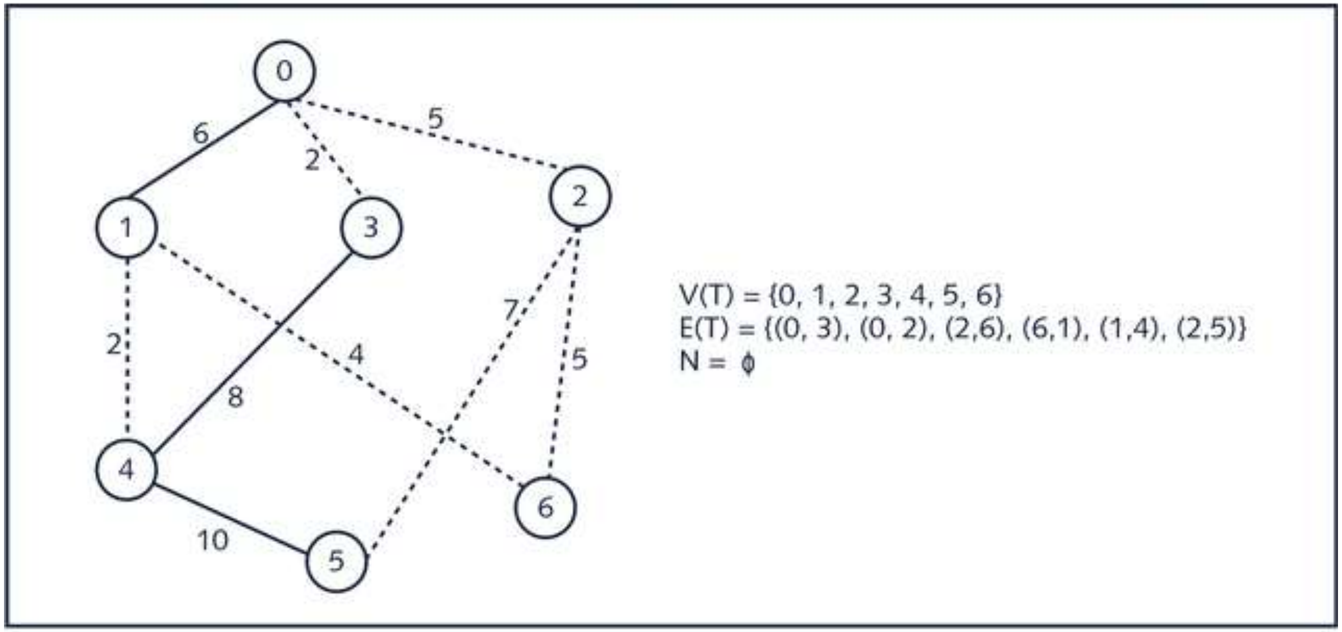
Prim’s Algorithm



V(T) = {0, 1, 2, 3, 4, 6}
E(T) = {(0, 3), (0, 2), (2,6), (6,1), (1,4)}
N = {5}

Figure 12-22 Graph *G*, *V*(*T*), *E*(*T*), and *N* after the fifth iteration of Step 3

Prim's Algorithm



V(T) = {0, 1, 2, 3, 4, 5, 6}
E(T) = {(0, 3), (0, 2), (2,6), (6,1), (1,4), (2,5)}
N = ϕ

Figure 12-23 Graph *G*, *V*(*T*), *E*(*T*), and *N* after the sixth iteration of Step 3

Applet

Spanning Tree As an ADT

```
template<class vType, int size>
class msTreeType: public graphType<vType, size>
{
public:
    void createSpanningGraph();
      //Function to create the graph and the weight matrix.
    void minimalSpanning(vType sVertex);
      //Function to create the edges of the minimal
      //spanning tree. The weight of the edges is also
      //saved in the array edgeWeights.
    void printTreeAndWeight();
      //Function to output the edges and the weight of the
      //minimal spanning tree.
protected:
    vType source;
    double weights[size][size];
    int edges[size];
    double edgeWeights[size];
};
```

Lecture 15

Topological Order

- Let G be a directed graph and $V(G) = \{v_1, v_2, \ldots, v_n\}$, where $n > 0$.
- A **topological ordering** of $V(G)$ is a linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that if v_{ij} is a predecessor of v_{ik}, $j \neq k$, $1 <= j <= n$, and $1 <= k <= n$, then v_{ij} precedes v_{ik}, that is, $j < k$ in this linear ordering.

Topological Order

- Because the graph has no cycles:
  - There exists a vertex *u* in *G* such that *u* has no predecessor.
  - There exists a vertex *v* in *G* such that *v* has no successor.

Topological Order

```
template<class vType, int size>
class topologicalOrderT: public graphType<vType, size>
{
public:
    void bfTopOrder();
      //Function to output the vertices in breadth first
      //topological order
};
```

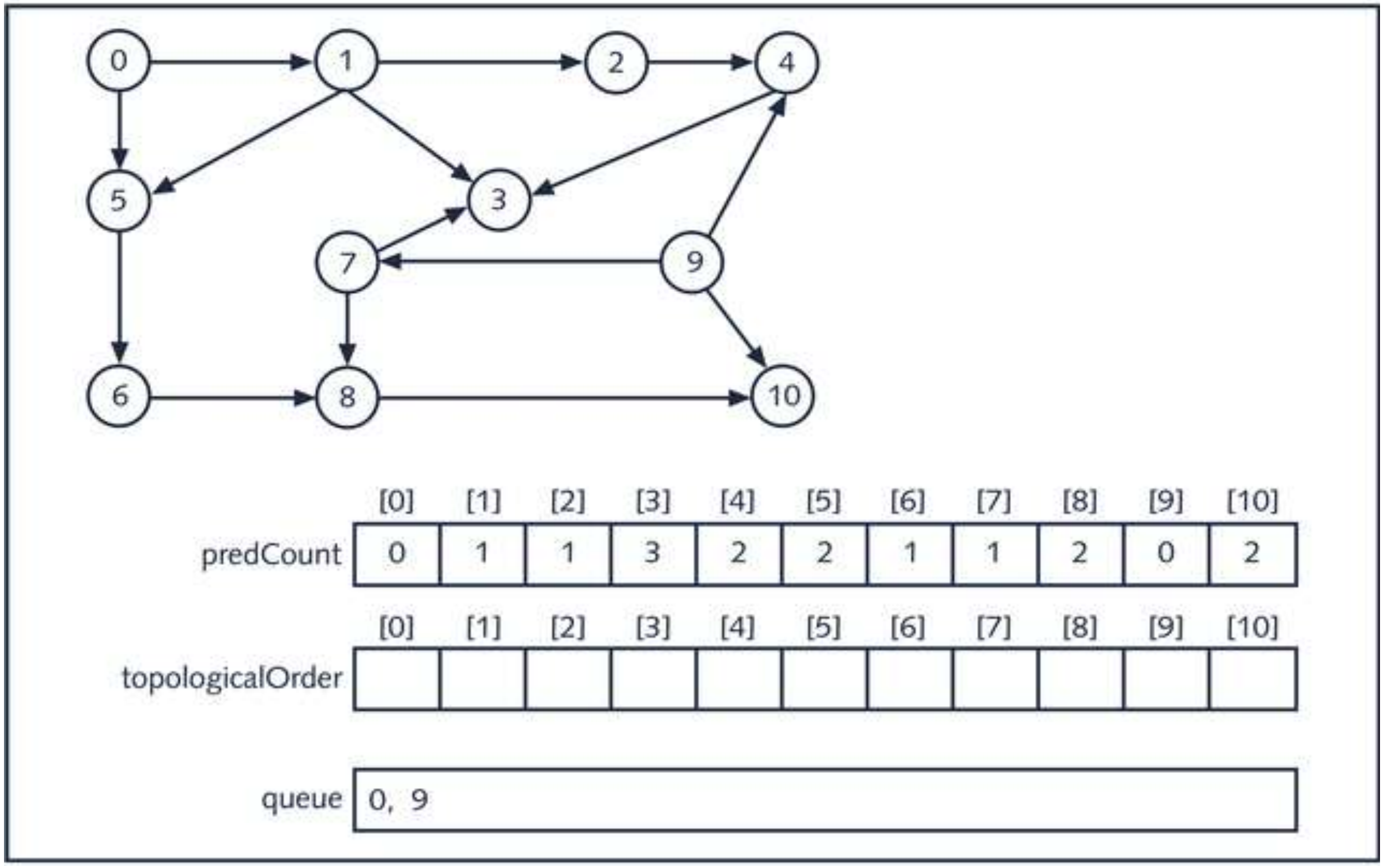
Breadth First Topological Order

1. Create the array predCount and initialize it so that predCount[i] is the number of predecessors of the vertex vi
2. Initialize the queue, say queue, to all those vertices vk so that predCount[k] is zero. (Clearly, queue is not empty because the graph has no cycles.)

Breadth First Topological Order

3. while the queue is not empty
    1. Remove the front element, u, of the queue
    2. Put u in the next available position, say topologicalOrder[topIndex], and increment topIndex
    3. For all the immediate successors w of u
        1. Decrement the predecessor count of w by 1
        2. if the predecessor count of w is zero, add w to queue

Breadth First Topological Order



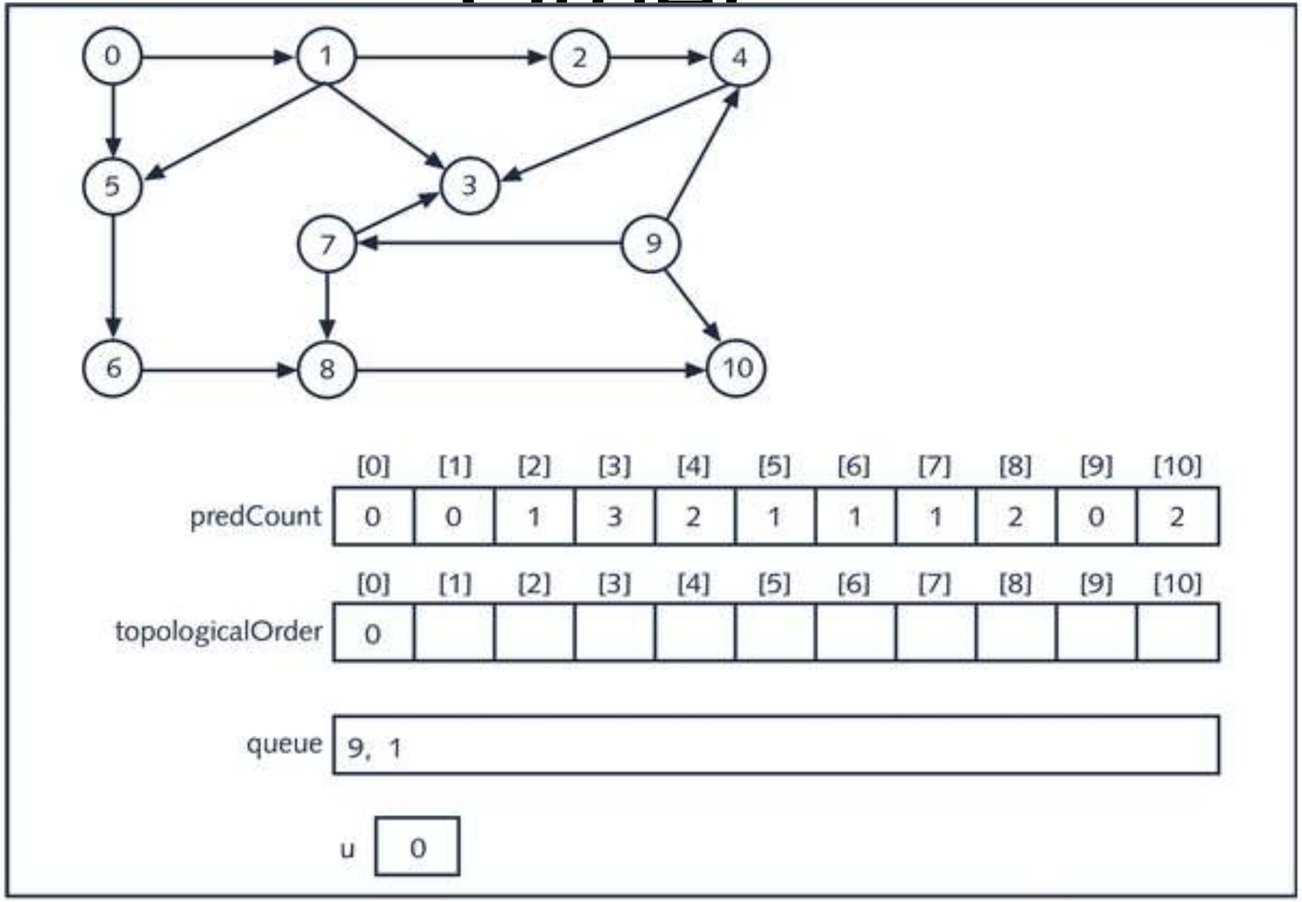
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
predCount	0	1	1	3	2	2	1	1	2	0	2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
topologicalOrder											

queue: 0, 9

Figure 12-24 Arrays `predCount`, `topologicalOrder`, and `queue` after Steps 1 and 2 execute

Breadth First Topological Order



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
predCount	0	0	1	3	2	1	1	1	2	0	2

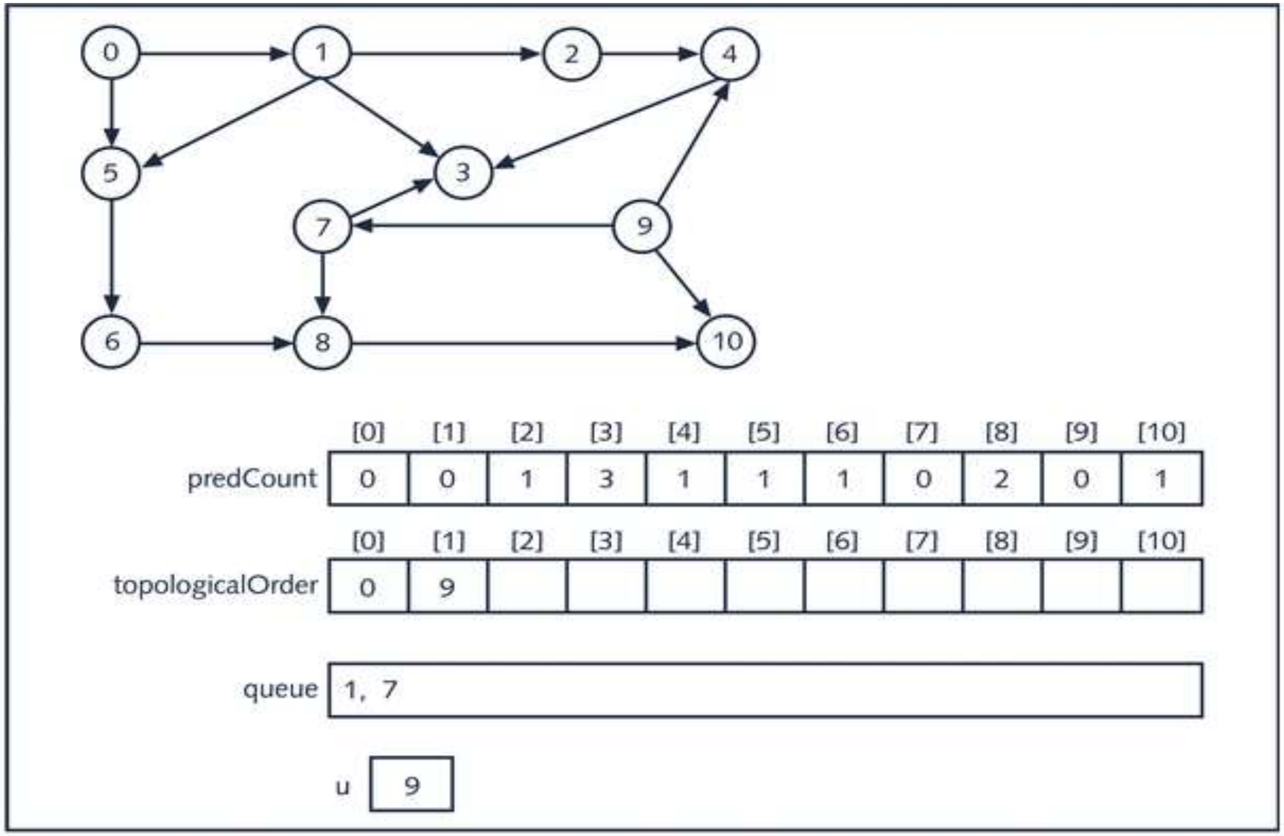
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
topologicalOrder	0										

queue: 9, 1

u: 0

Figure 12-25 Arrays `predCount`, `topologicalOrder`, and `queue` after the first iteration of Step 3

Breadth First Topological Order



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
predCount	0	0	1	3	1	1	1	0	2	0	1

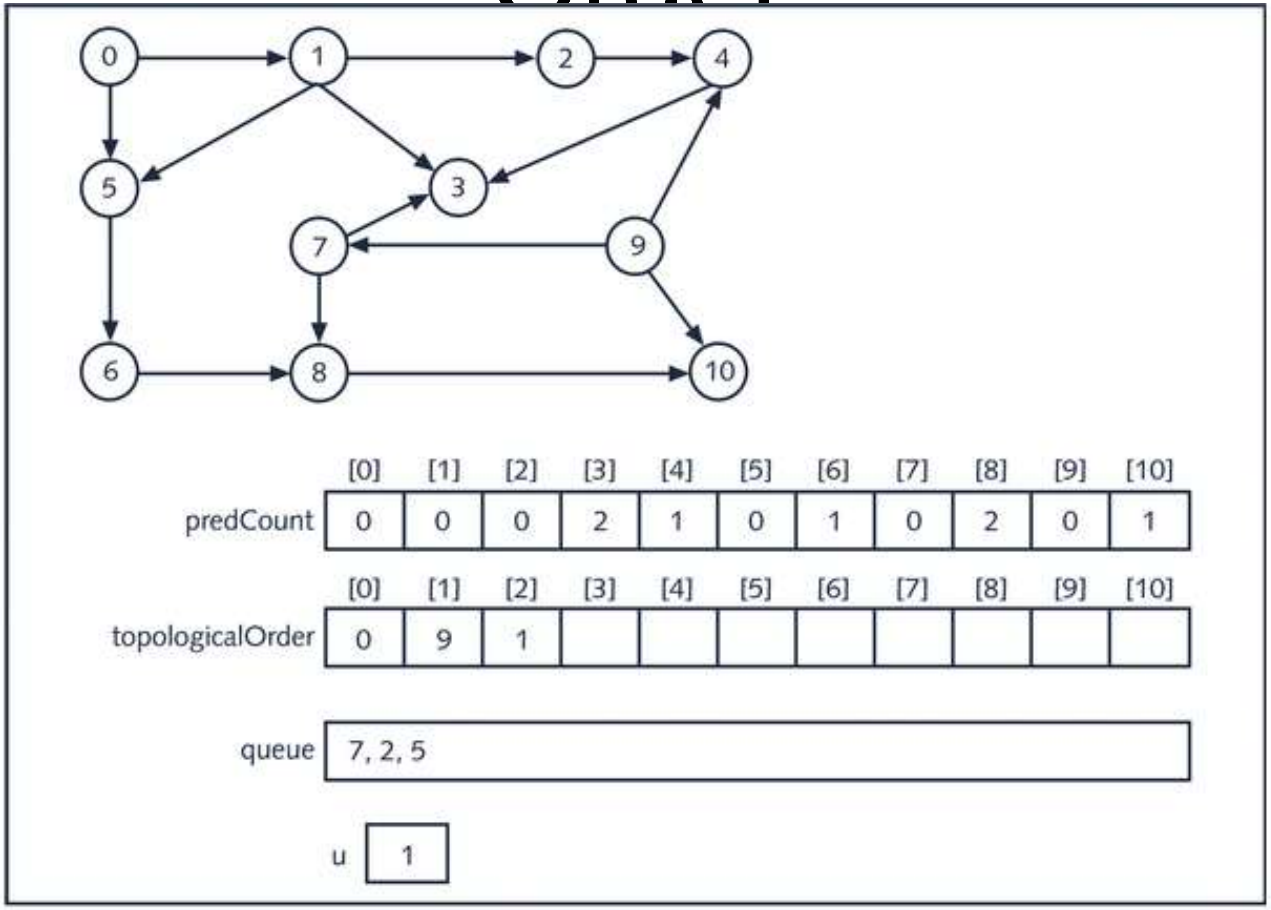
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
topologicalOrder	0	9									

queue 1, 7

u 9

Figure 12-26 Arrays `predCount`, `topologicalOrder`, and `queue` after the second iteration of Step 3

Breadth First Topological Order



predCount

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	0	0	2	1	0	1	0	2	0	1

topologicalOrder

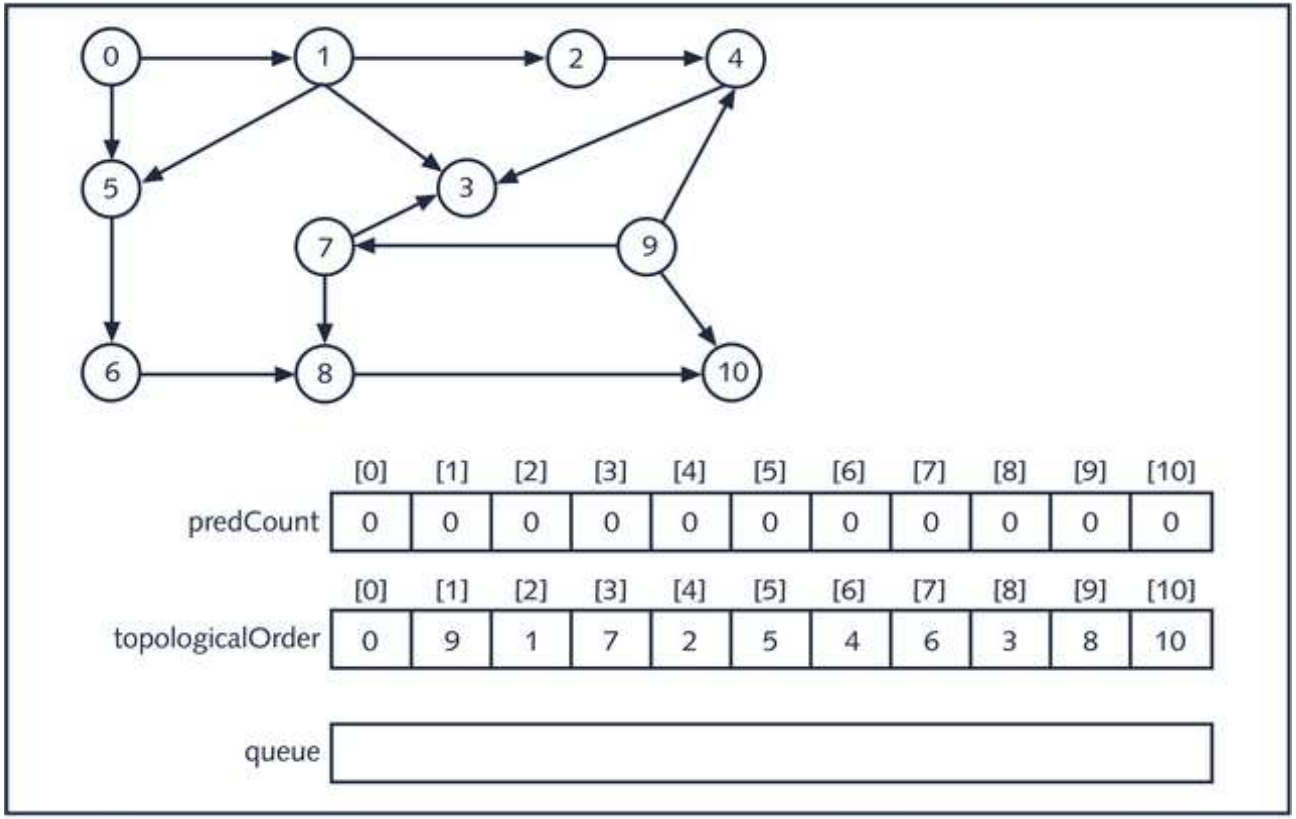
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
0	9	1								

queue: 7, 2, 5

u: 1

Figure 12-27 Arrays `predCount`, `topologicalOrder`, and `queue` after the third iteration of Step 3

Breadth First Topological Order



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
predCount	0	0	0	0	0	0	0	0	0	0	0

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
topologicalOrder	0	9	1	7	2	5	4	6	3	8	10

queue

Figure 12-28 Arrays `predCount`, `topologicalOrder`, and queue after Step 3 executes eight more times