Data Structures

Data Modelling/Data Structuring

- Representation of data elements and relationship between them
- Data Structure: data representation and its associated operations
- Examples: Integers, float point numbers
- A data structure meant to be an organization or structuring for a collection of data items
- Example: An ordered list of integers stored in an array
  - Search
  - Print and process
  - Modification

Data Structures

- Data Structure: A scheme/particular way of organizing data in the memory of a computer
- Different kinds of data structures are suited for different applications
  - For databases B-Trees
  - For Compilers Hash Tables
- Data structures are used in almost every program or software systems
- These are essential components of many efficient algorithms and make management of huge amounts of data possible
  - Large databases
  - Internet indexing

Data Structures

- The logical or mathematical model of a particular organization of data is called a data structure
- A data model/data structure depends on two things:
  - It must mirror the actual relationship of the data items in the real world
  - It must be simple to process the data when necessary

Motivation

- Consider a program that reads a triangle and outputs the area
- No variable of type triangle can be defined and read
- This program needs new data structures, since there is no data structure for triangle
- A triangle is a set of three points in the plane and a point in the plane can be represented by its x and y coordinates
- Declare a structure for point which consists of x and y coordinates of a point as its fields
- Triangle is a structure that consists of three fields for its three corners
- Using built-in types defined more complex types

Motivation

- There are no functions to read in points and triangles
- Define functions for reading in points and triangles
- There is no function for computing area of a triangle
- Define a function for the same
- Using built-in types we may have to write more complicated types or structures according to the requirement
- If required, we may write functions for printing points and triangles

Motivation

- While solving a programming problem, we need to identify the data structures and algorithms that are required to solve the problem
- In the triangle area problem, two new data structures and an algorithm to find the area of the triangle are required
- Different data structures may be defined to solve the same problem and different implementations are possible for the same data structure or algorithm
- For example, we can use “r” theta representation to represent point
- A triangle can be represented by its sides instead of its corners
- Using the data structure triangle, we can write many other functions that solve various problems related to triangles

Selection of a data structure

- Nature of the data and the operations/processes that need to be performed on the data decide the selection
- The steps to be follow while selecting a data structure are:
  - Problem analysis and identification of the operations that need to be supported
  - Quantify of resource constraints for each operation
  - Select the data structure that meets these requirements in the best possible way

Selection of a data structure

- Examples of basic operations: inserting a data item into a data structure, deleting a data item from a data structure, and finding a specific data item
- Resource constraints on certain key operations such as search, inserting data records, and deletion data records, drive the data structure selection process
- Many issues relating to relative importance of these operations are addressed by the following questions:
  - Are all data items are inserted into the data structure at the beginning or are the insertions interspersed with other operations?
  - Can data items be deleted?
  - Are all data items processed in some well-defined order, or is search for specific items allowed?

Overview

- The study of a data structure consists of three steps:
  1. Logical or mathematical description of the structure
  2. Implementation of the structure on a computer
  3. Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure and perform the required operations

Overview

- Linear data structures such as: Lists, restricted access lists (stacks and queues)
- Non-linear data structures: Trees, heaps, tries, graphs
- Dictionaries: Sorting, searching, hashing, hash tables, Bloom filters

Overview

- Implementation issues:
  - recursive and iterative implementation, dynamic allocation
  - Ex: Arrays or linked lists

Overview

- Performance analysis: A study of efficiency of a data structure and an algorithm
- Efficiency in terms of space analysis and time analysis
- Efficiency of algorithms as a function of the input size
  - The running will increases with the size of the input
  - Study the growth of the running time with the input size
- How to measure the running time?
  - Can be done using an experimental study
  - Implement your algorithm in certain platform and measure its running time (using a system clock) on different types of inputs
  - Based on this analysis make conclusions about the running time

Overview

- There are few limitations
  - Have to implement the algorithm
  - Study can only be done on the subset of input, which may not give a proper indication of the running time of your algorithm
  - To make a choice between two algorithms, your experimental study should be based on the same platform
- It is advantages to have a general methodology to analyse the running time of an algorithm
- How this methodology should work?
  - Take a high level description (pseudo-code) of an algorithm
  - Take all possible inputs into consideration
  - Allow the analysis independent of the software and hardware platform

Overview: Pseudo-code

- What is pseudo-code?
  - A mixture of natural language and high-level programming concepts that describe the main ideas behind a general implementation of any data structure or algorithm
  - Example:

```
Algorithm arrayMax(A, n)
Input: An array A storing n integers and the size
Output: The maximum element in A
currentMax = A[0];
for i=1 to n-1 do
        if currentMax < A[i] then
                currentMax = A[i];
return currentMax
```

Abstraction

- An abstract data type is a logical description of how we view the data and the set of operations allowed on that data without regard to how they will be implemented
- An abstract data type (ADT) is a set of objects together with a set of operations
- It is a method to bring flexibility to your logical data model
- We are concerned with what the data is representing and not with how it will eventually be constructed
- With this level of abstraction we are creating an encapsulation around the data
- By encapsulating implementation details, we are hiding them from the user's view

Abstraction

- To implementation an ADT (also referred as a data structure), we have to provide a physical view of the data using a collection of programming constructs and primitive data types
- We provide a separation between logical view and physical view of the data
- Provides implementation-independent view of the data
- Provides freedom to the programmer to switch the implementation details without changing the way interaction for the data
- The user can focus on the problem solving process

Abstraction

- The implementation details of the methods are not mentioned in an ADT’s definition
- Objects such as lists, sets, and graphs along with their operations can be viewed as ADTs
- Operations to be supported by an ADT is a design decision

Data Structures

List Abstract Data Type

- A container of elements that stores each element at a particular position and keeps these positions arranged in a linear order
- It is a general list of the form A0, A1, . . . , Ai-1, Ai, Ai+1, . . ., An-1
- Position is always defined relatively
- Position “p” is always after some position “q” and before some position “s” unless “p” is either first or last position
- Position “p” does not change if we swap or replace the element “e” stored at p with another element
- In the list A0, A1, A2, . . ., An-1, the position of Ai is “i”
- The size is n
- List of size 0 is called an empty list
- Ai follows Ai-1 (i<n) and Ai-1 precedes Ai (i>0)
- A0 is the first element and An-1 is the last element

Lists

- Some of the methods supported by the list ADT are:
- first()
- last()
- isFirst(p)
- isLast(p)
- replaceElement(p, e)
- swapElements(p, q)
- insertFirst(e)
- insertLast(e)
- remove(e)
- removeAtPosition(p)
- Find(e)
- insert(e, p)

Lists: Example

- Example list: 2, 56, 4, 34, 1, 5, 16, 12, 23, 10
- replaceElement(1, 55)
  - 2, 55, 4, 34, 1, 5, 16, 12, 23, 10
- swapElements(2, 3)
  - 2, 56, 34, 4, 1, 5, 16, 12, 23, 10
- insertFirst(22)
  - 22, 2, 56, 4, 34, 1, 5, 16, 12, 23, 10
- insertLast(24)
  - 2, 56, 4, 34, 1, 5, 16, 12, 23, 10, 24
- remove(4)
  - 2, 56, 34, 1, 5, 16, 12, 23, 10
- find(5)
  - Return 5
- Insert(4, 25)
  - 2, 56, 4, 34, 25, 1, 5, 16, 12, 23, 10

Implementation using an array

- A list can be implemented with an N-element array S
- Elements stored from S[0] to S[n-1]; S[0]: the first element and S[n-1] is the last element
- How to insert a new element when the size is N?

Algorithm insert(e, p)

for i=n-1, n-2, . . ., p do

S[i+1] ← S[i] {Make room for “e” to be inserted}

S[p] ← e

n ← n+1

Implementation using an array

Algorithm removeAtPosition(p)

e ← S[p] {e is a temporary variable}

for i=p, p+1, . . ., n-2 do

S[i] = S[i+1] {fill in for the remove element}

n ←n-1

Implementation of Lists

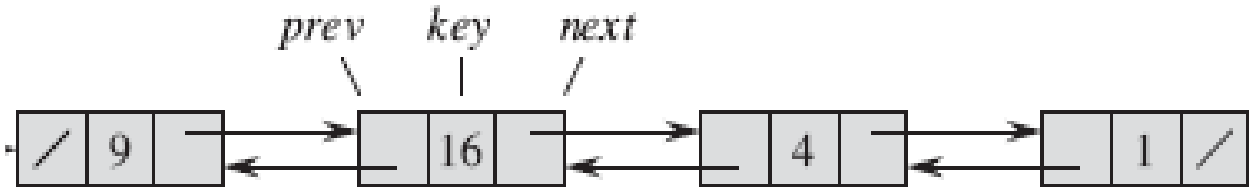
- All operations can be implemented using an array
- Insertion and deletion are expensive, depending upon where the insertions and deletions occur
- Insertion at position “0”
- Deletion of the first element

Implementation of Lists

- A linked list is a data structure where the objects are arranged in a linear order
- In array based implementation, linear order is determined by array indices
- In linked list based implementation, linear order is determined by the pointer or link maintained in each object

Doubly linked list

- This is an example of a doubly linked list



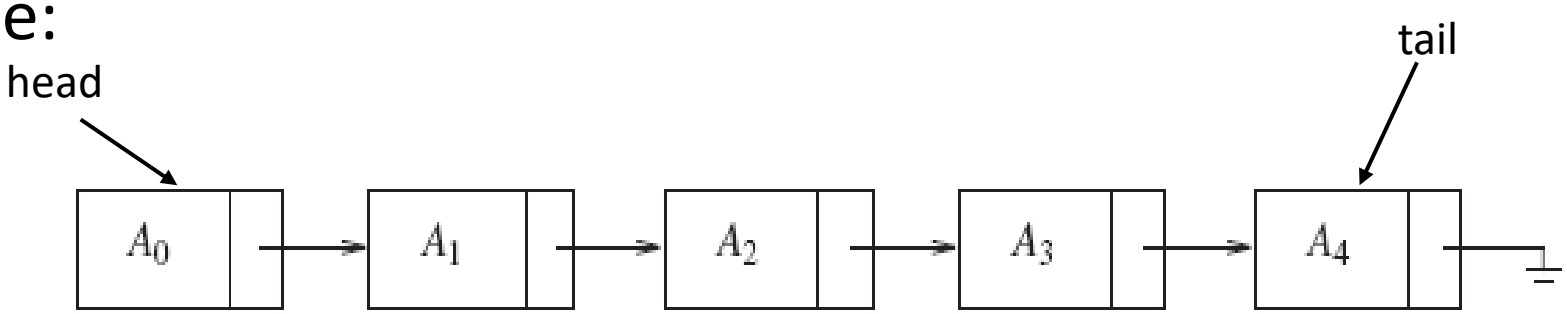
- Each node/object of a doubly linked list stores a “key/element” and two links/references called “next” and “prev”
- The “next” link points to the next node in the list and the “prev” link points to the previous node in the list
- Can be traversed in either direction
- The “next” and “prev” links of a node
- If the “next” or “prev” links of a node are “NIL”, then?

Variants of a linked list

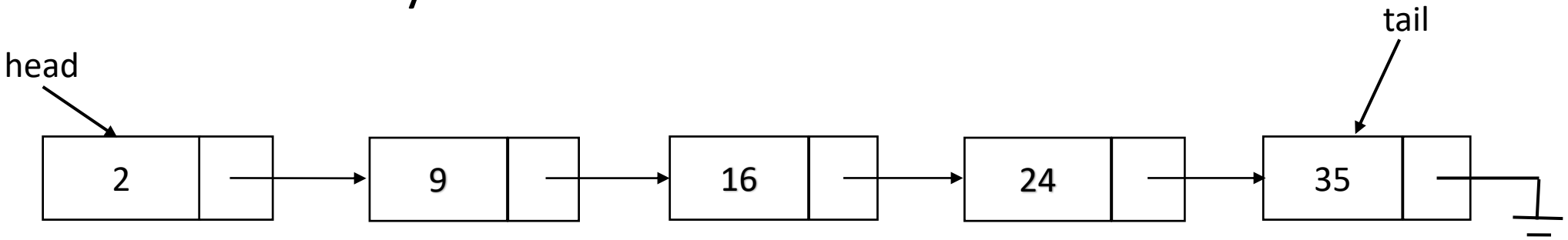
- A list may have one of the several forms:
  - Singly linked or doubly linked
  - Sorted or not
  - Circular or not
  - Lists with sentinels

Singly linked list

- If a list is singly linked, then each node have a link/reference to only the next node in the list
- Example:

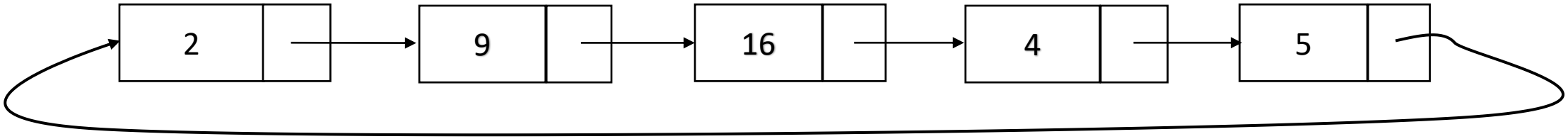


- If a list is sorted, the linear order of the list corresponds to the sorted order of the keys stored in nodes

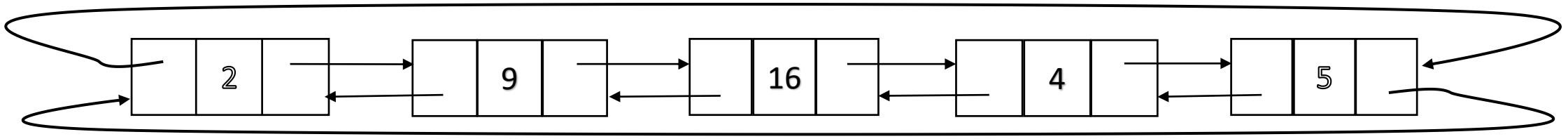


Circular singly linked list

- If the next link of the last node points to the first node, then the resultant list is called a circular singly linked list



- Circular doubly linked list: Make the “next” link of the tail to point to the head and the “prev” link of the head to point to the tail

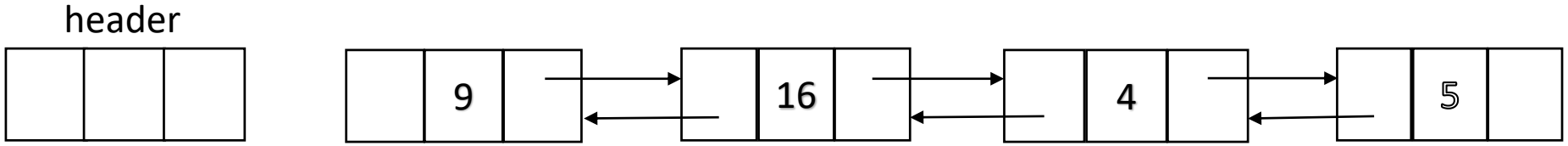


Linked lists with sentinels

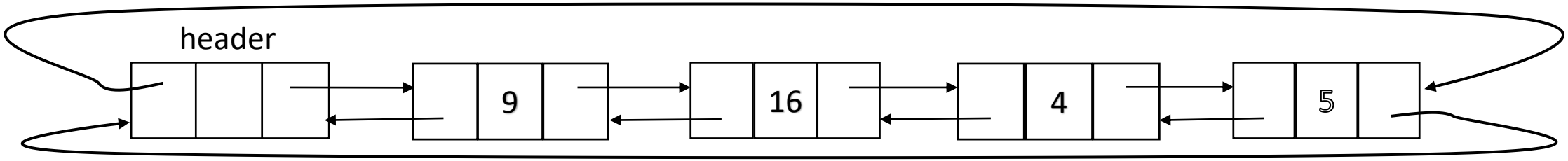
- To simplify search and update operations, it is convenient to add a special node at the beginning/ending of a list: a header node just before the head of the list; a trailer node after the tail
- A sentinel is a dummy node does not store any data element but has all valid links similar the other nodes

Linked lists with sentinels

- Consider a doubly linked list



- The “next” link of header points to the head of the list and the “prev” link points to the tail of the list
- Also make the “next” link of the tail to point to the header and the “prev” link of the head to point to the header
- The resultant list is called “circular doubly linked list with sentinel”



Implementation using linked lists

Algorithm insertAfter(p, e)

create a new node v

v.element = e

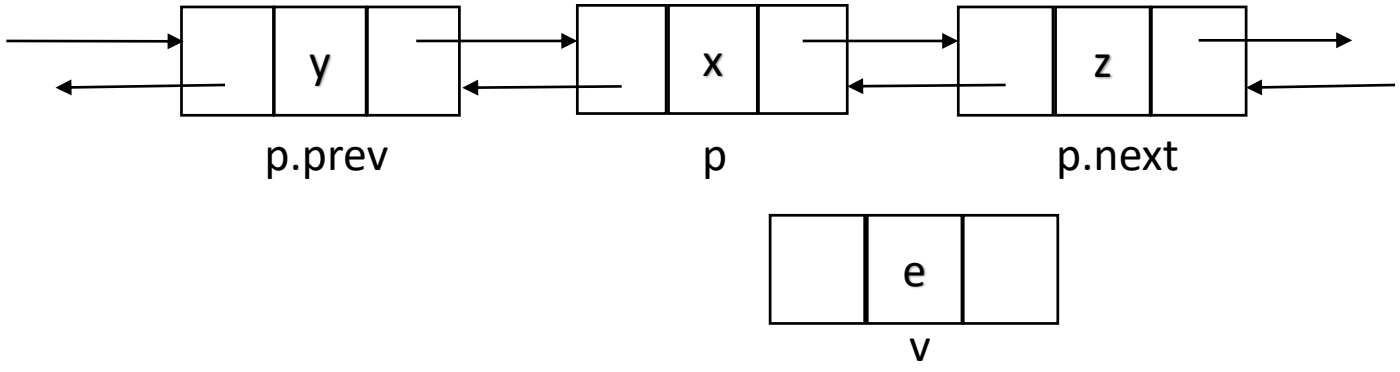
v.prev = p {link v to its predecessor}

v.next = p.next {link v to its successor}

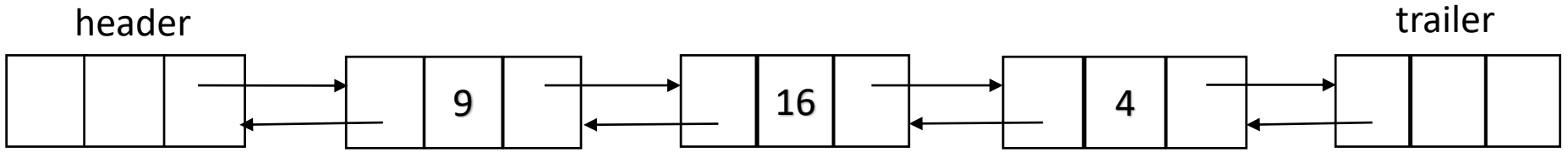
(p.next).prev = v {link p's old successor to v}

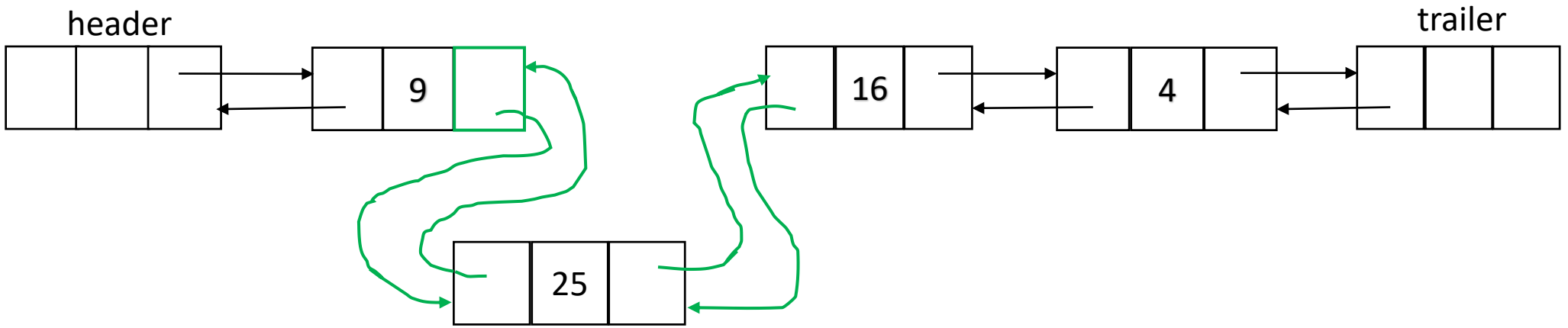
p.next = v {link p's old predecessor to v}

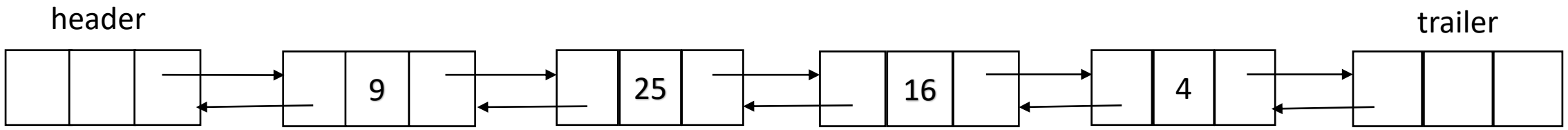
return v {the position of element "e"}



Insertion: Example







Implementation using linked lists

Algorithm removeAtPosition(p)

t ← p.element {a temporary variable to hold the return value}

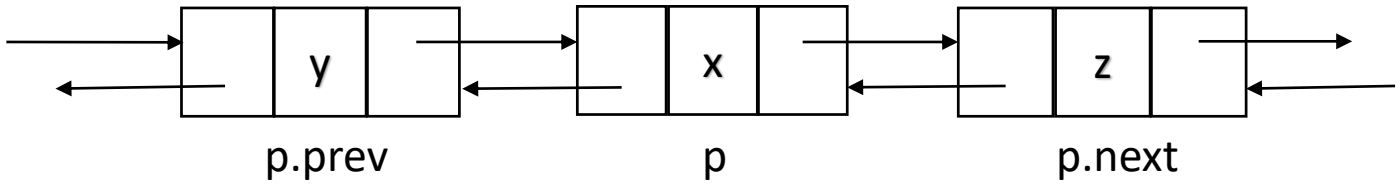
(p.prev).next = p.next {link out p}

(p.next).prev = p.prev {link out p}

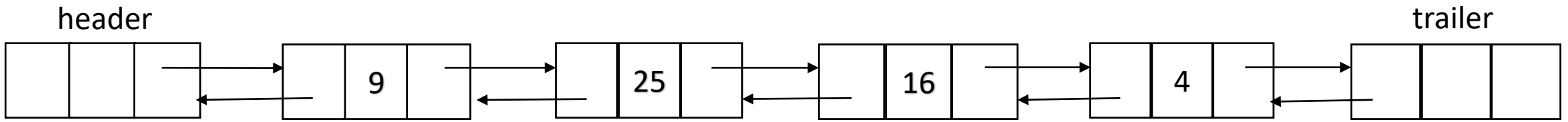
p.prev = null {invalidate the position p}

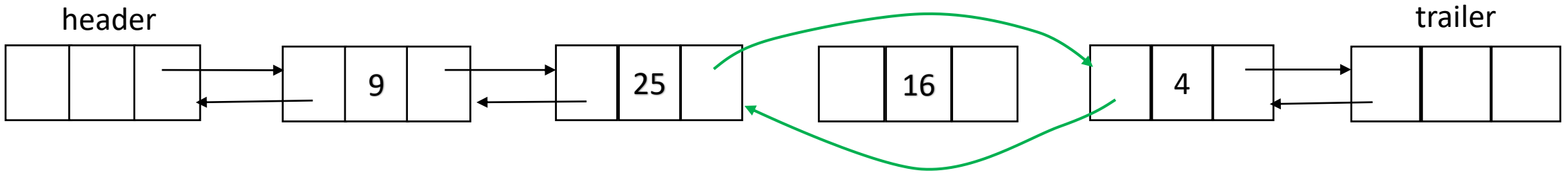
p.next = null

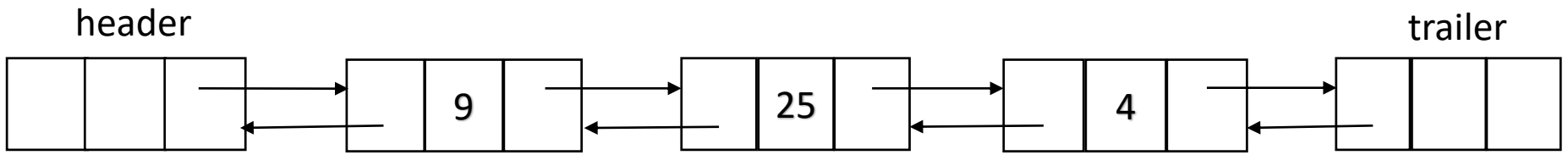
return t



Deletion: Example







Data Structures

Stack

- A container of data elements/objects that are stored and removed according to last in first out (LIFO) principle
- Objects are inserted at the top of stack and also removed from the top, that is, can remove the most recently inserted object at a time
- Insertion: pushing; deletion: popping

Stack

- Stack is an abstract data type that supports these operations:
  - push(e): inserts the object “e” at the top of the stack
  - pop(): removes and returns the top object from stack, if the stack is empty an error occurs

Stack

- The supporting operations of the stack ADT are:
  - size() – returns the number of elements in the stack
  - isEmpty() – Tells us whether there are any objects in the stack or not
  - top() - returns the top object of the stack, without removing the object, if the stack is empty an error occurs

Implementation using an array

- A stack can be easily implemented with an N-element array S
- The elements are stored from S[0] to S[t], “t” is the index of the topmost element in the stack

Implementation using an array

- Assume that array index starts from “0”, and initialize t to “-1”
- The value of t is used to identify when the stack is empty, and the size of the stack
- An exception must be raised when the stack becomes full

Algorithm push(o)

```
    if size()  = N then
        indicate a stack-full error has occurred
    t ←t+1
    S[t] = o
```

Implementation using an array

```
Algorithm pop()
        if(isEmpty()) then
                indicate a stack-empty error has occurred
        e ← S[t]  {e is a temporary variable}
        S[t] ← null
        t ← t-1
        return e
```

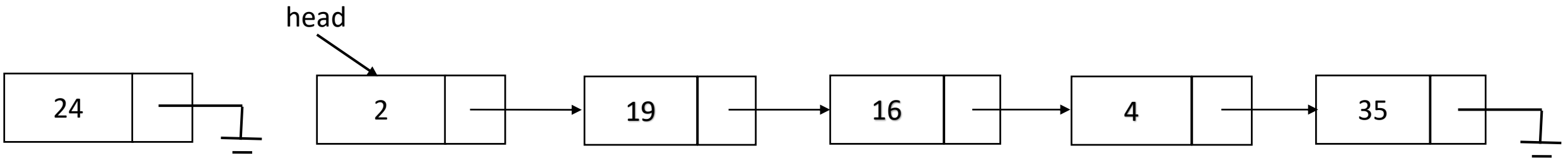
Implementation using an array

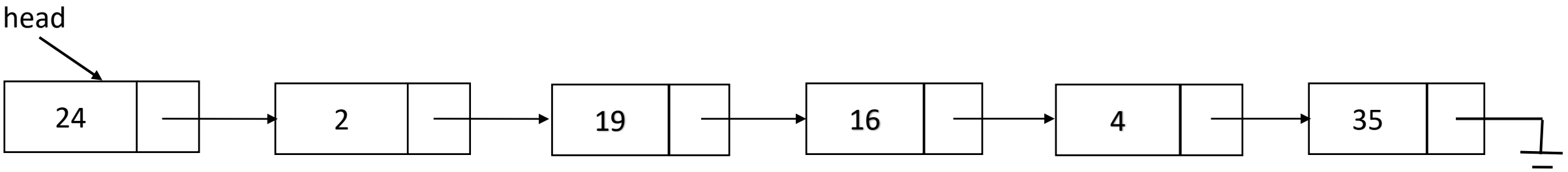
```
Algorithm push(e)
        if size() == N then
                A←new array of length f(N)
        for i ←0 to N-1
                A[i] ← S[i]
        S ←A
        t ←t+1
        S[t] ← e
```

- tight strategy: f(N) = N+c
- Growth strategy: f(N) = 2N

Implementation using a linked list

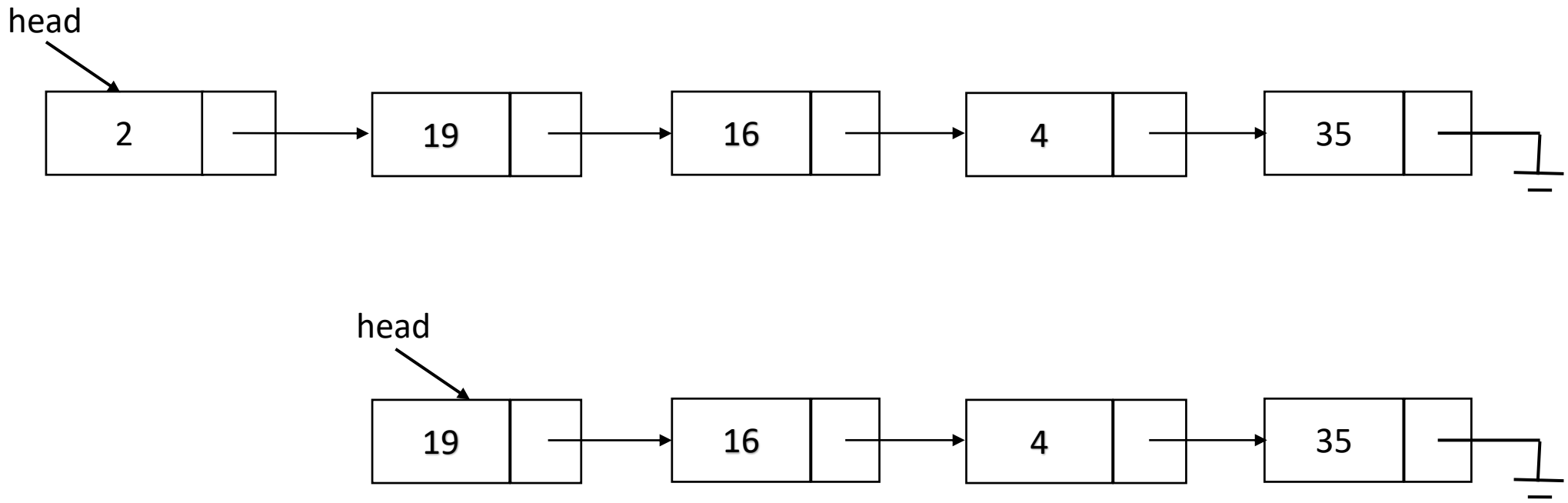
- push operation
  - Create a new node (temp) with the data element to be inserted
  - Update the next line of temp to point to the node referred by “head”
  - Update the head to refer to temp





Implementation using a linked list

- Pop operation
  - Update head to refer to next node of top of stack
  - Update the next link of top of stack to refer to null; free the memory allocated to deleted node



Implementation using a linked list

- Some of the methods implemented as a part of linked list ADT are:

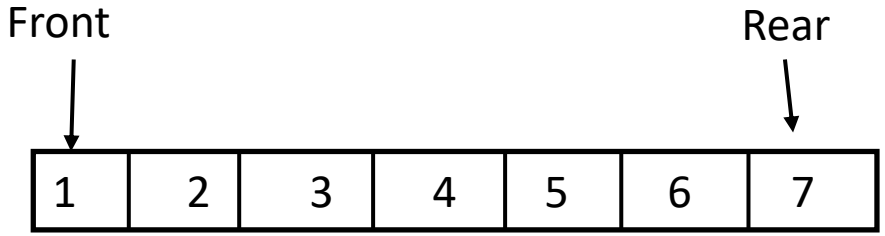
  first(), insertAtPosition(e,p), remove(p), retrieve(p)

Array or linked list based implementation

- Must assume a fixed upper bound on the ultimate size of the stack
- Waste of memory
- Linked lists:
- Do not have size limitation
- Uses space in proportion to the number of elements in the stack

Queue

- A queue is container of data elements/objects that are inserted and removed according to the first-in-first-out (FIFO) principle
- Elements can be inserted at any point of time
- Can only remove the element which has been there for the longest
- Elements enter the queue at the “rear” and removed from the “front”



Queue: ADT

- Keeps objects in a sequence
- Access and deletion is limited to the first (front) element in the sequence
- Insertion is restricted to the end (rear) of the sequence

Fundamental methods:

- enqueue(o)
- dequeue()

Supporting methods:

- size()
- isEmpty()
- front()

Implementation using an array

- Define an array Q of size N
- Define two variables “f” and “r” to enforce FIFO principle

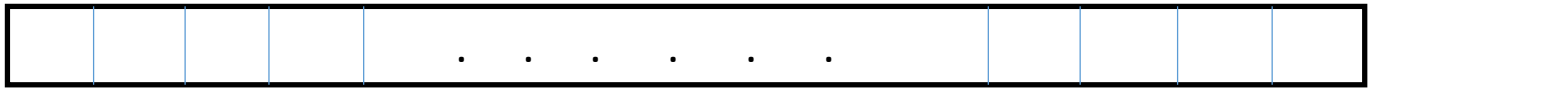
  f: index to the cell of Q storing the first element of the queue

  r: index to the next available array cell of Q
- f = r = 0
- If f = r indicate that the queue is empty
- Increment f when an element is removed from the queue
- Increment r when an element is inserted into the queue

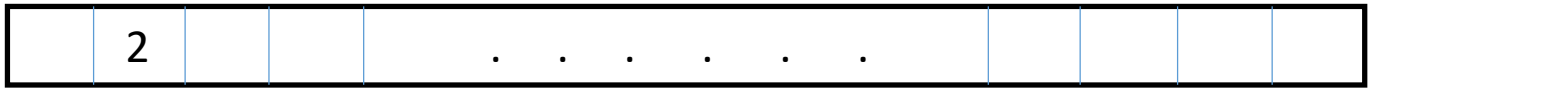
Implementation using an array

- Take an empty queue
- Insert an element and remove it; repeat this cycle for N times

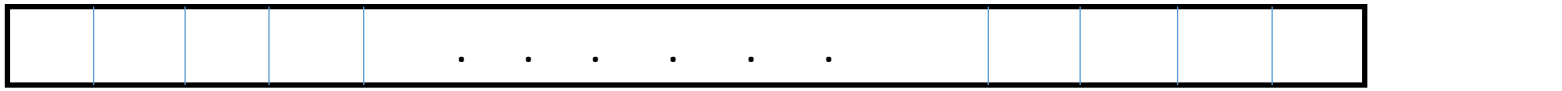
f r



f, r



f r

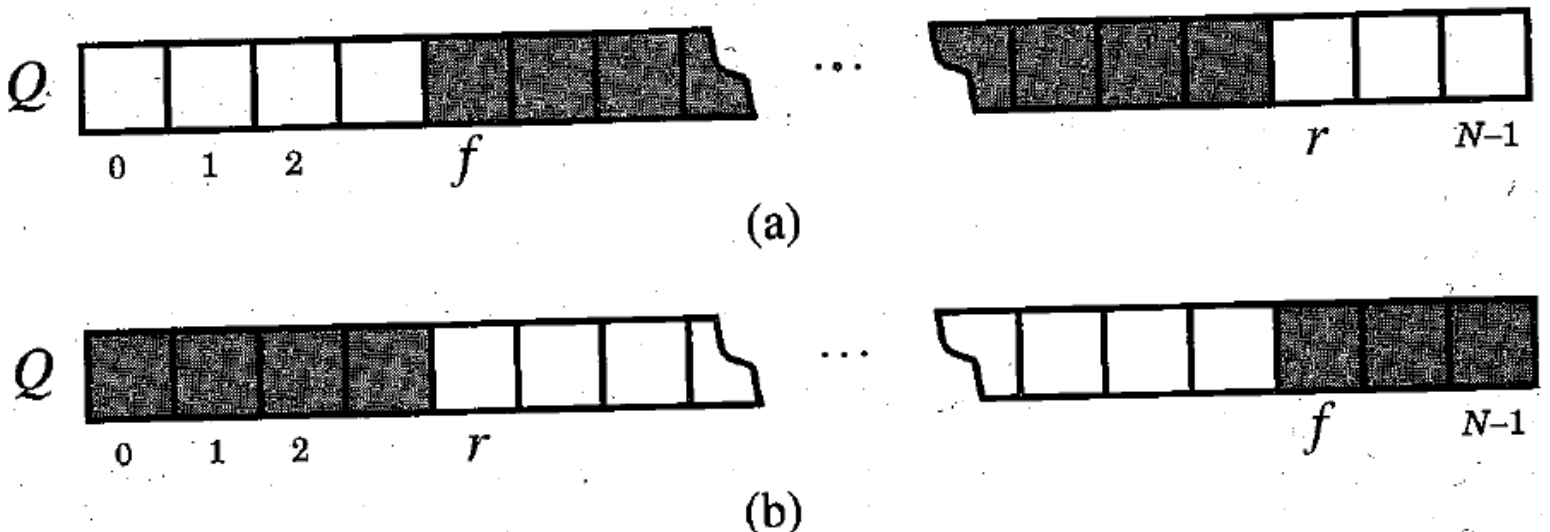


f, r

f, r

Implementation using an array

- Insertion results in array-out-of-bounds error
- Not able to insert in spite of plenty empty cells
- Let “f” and “r” wrap around the queue
- View Q as a circular array
- f = (f+1) mod N; r = (r+1) mod N



Implementation using an array

- Consider the scenario, enqueue N elements one-by-one
- f=r
- Ambiguity in distinguishing between an empty and a full queue
- Do not let the queue to hold more than N-1 elements

Implementation using an array

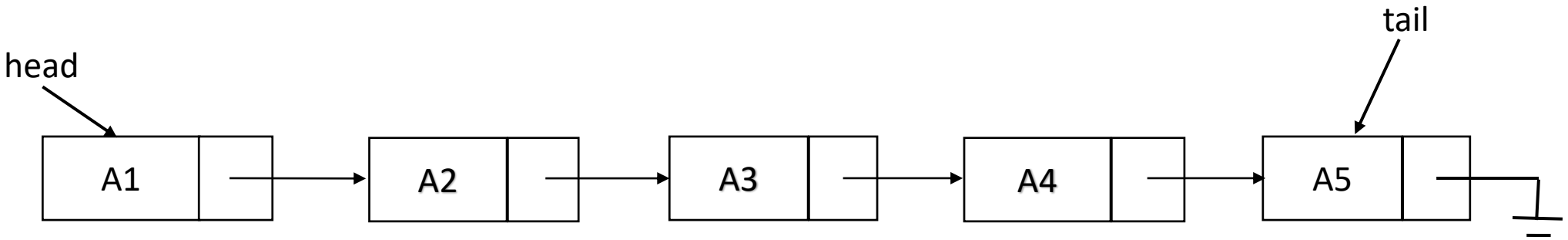
```
Algorithm enqueue(e)
        if (size() == N-1)
                raise QueueFull exception
        Q[r] = e
        r ← (r+1) mod N
```

Implementation using an array

```
Algorithm dequeue()
        if isEmpty() then
                raise QueueEmpty Exception
        temp ← Q[f]
        Q[f] = null
        f ← (f+1) mod N
        return temp
```

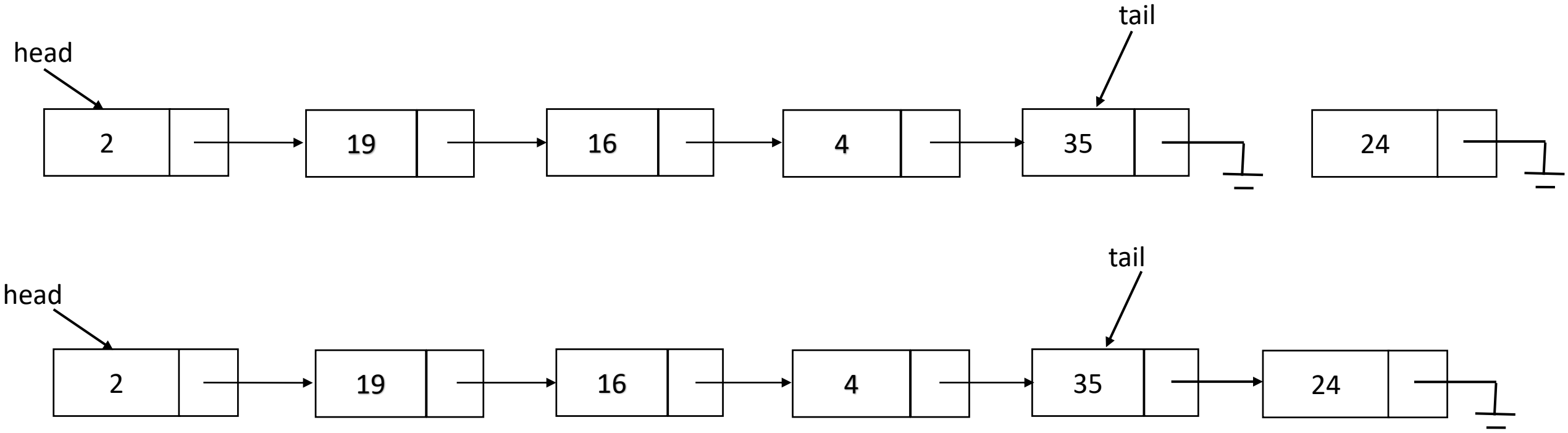
Implementation using a linked list

- What should be the front of the queue, head or tail
- To reduce the overhead, head should be the front of the queue



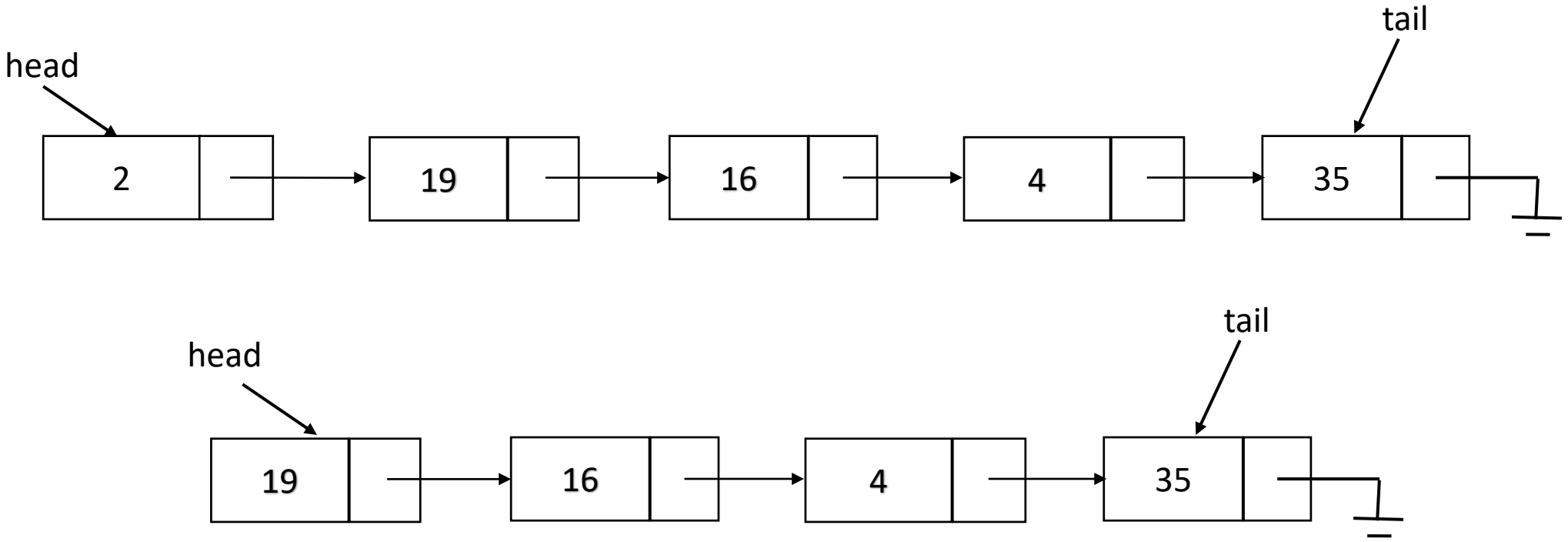
Implementation using a linked list

enqueue



Implementation using a linked list

- dequeue



Dynamic Sets

- Collection of objects whose size may change
- These collections of elements are called dynamic sets
- Lists, stacks and queues

Data Structures

Sorted or unsorted lists

- Make a choice between sorted or unsorted implementation
- Depends upon the operations to be supported by the list and relative importance of these operations
- To search for an element in a sorted list, we can use binary search

  Ex: A = {2, 3, 6, 8, 12, 23, 33, 45, 65}

  We want to search “65”

  Initialize L=0 and R = 8

  compute m = floor((L+R)/2)=4; Since 12 < 65, update L as: L = 5

  compute m = floor((L+R)/2)=6; Since 33 < 65, update L as: L = 7

  compute m = floor((L+R)/2)=7; Since 45 < 65, update L as: L = 8

  compute m = floor((L+R)/2)=8; since 65 found at position 8 return the same

Sorted or unsorted lists

- To search for an element in an unsorted list, we have to use sequential search

  Ex: A = {2, 3, 6, 8, 12, 23, 33, 45, 65}

  We want to search “65”

- Similar is the case with deletion operation
- Depending upon the relative importance of the operations we can make a choice between sorted and unsorted lists

Analysing Algorithms

- Analysing the dependency of running time on the size of input
- A general methodology that associates a function f(n) to characterize the running time in terms of input
  - A language for describing algorithms
  - A computational model that algorithms execute within
  - A metric for measuring algorithm running time
  - An approach for characterizing running times

Pseudo-code

- The programming language constructs that will be used:
  - Use standard mathematical symbols to describe numeric and Boolean operations/expressions
  - Use “←” for assignment instead of “=“
  - Use “=“ for equality relationship
  - Method declaration: Algorithm name(param1, param2)
  - Decision structures: if … then … [else …]
  - while-loops: while . . . do
  - for-loops: for . . . do
  - Array indexing: A[i], A[i,j]
  - Method calls: object.method(args)
  - Method return: return value
  - Use indentation to signify the beginning of a new loop

Analytic approach

- Define a set of high-level primitive operations independent of programming language
  - Data movement (assign)
  - Switching control (branch, subroutine call, return)
  - Logical and arithmetic operations (addition, comparison)
  - Indexing into an array
- The execution time of these primitive operations is dependent on the hardware and software environment (constant)
- Count the number of primitive operations executed by the algorithm and use that count as a high-level estimate of the running time

Count the primitive operations

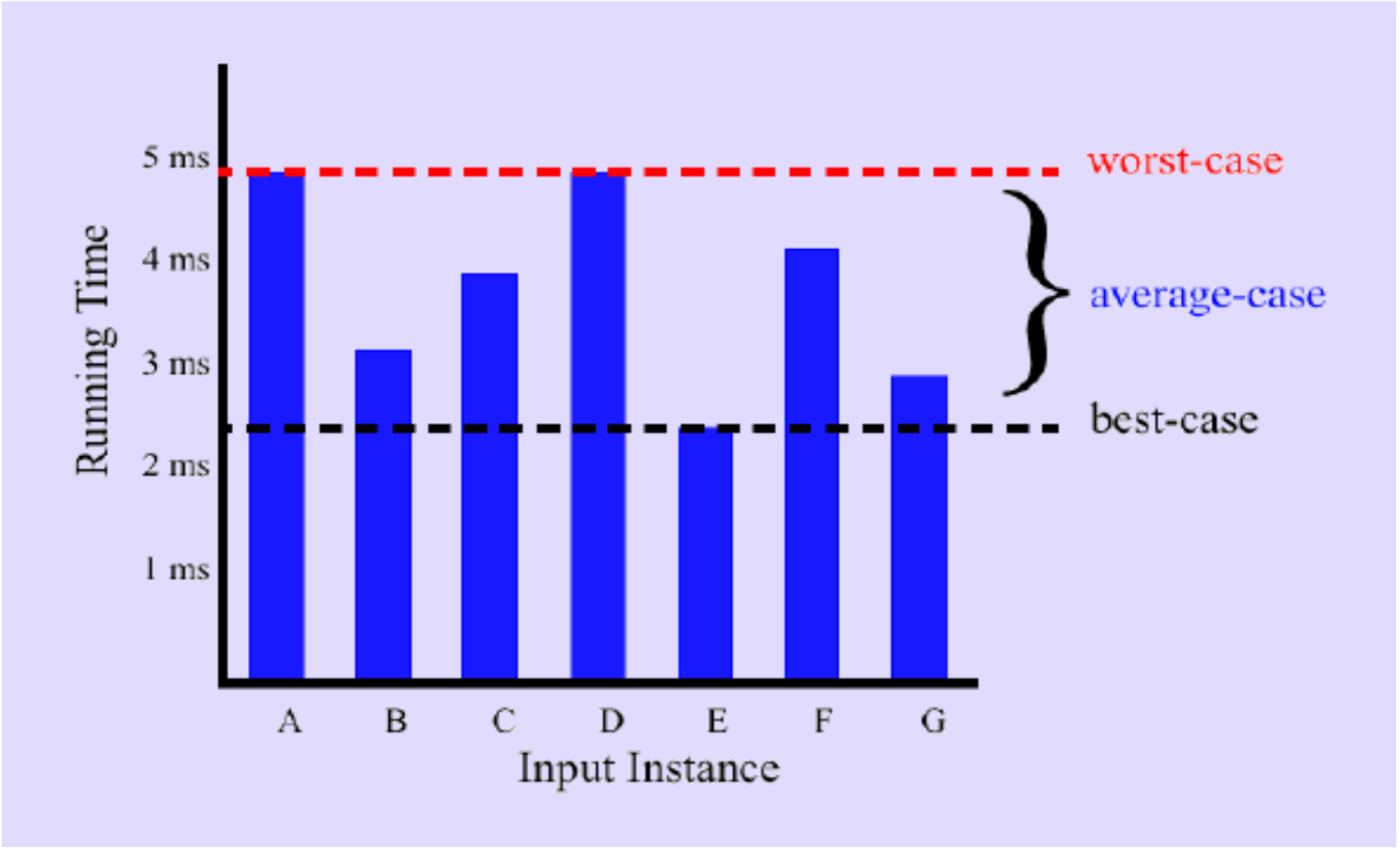
```
Algorithm arrayMax(A, n)
    Input: An array A storing n integers and the size
    Output: The maximum element in A
    currentMax ← A[0]                          2 units
    for i←1 to n-1 do                          1 + n units
            if currentMax < A[i] then          2(n-1) units
                    currentMax ← A[i]          0-2(n-1) units
                                               2(n-1) units
    return currentMax                          1 unit
```

Count the primitive operations

- Total running time of arrayMax is:
  - Best case: elements are in sorted decreasing order: 2+1+n+4(n-1)+1 = 5n
  - Worst case: elements are sorted increasing order: 2+1+n+6(n-1)+1 = (7n-2)
  - Average case: elements are partially sorted: between 5n and (7n-2)

Best, average, and worst case

- Expected running time based on a given input distribution



Best, average, and worst case

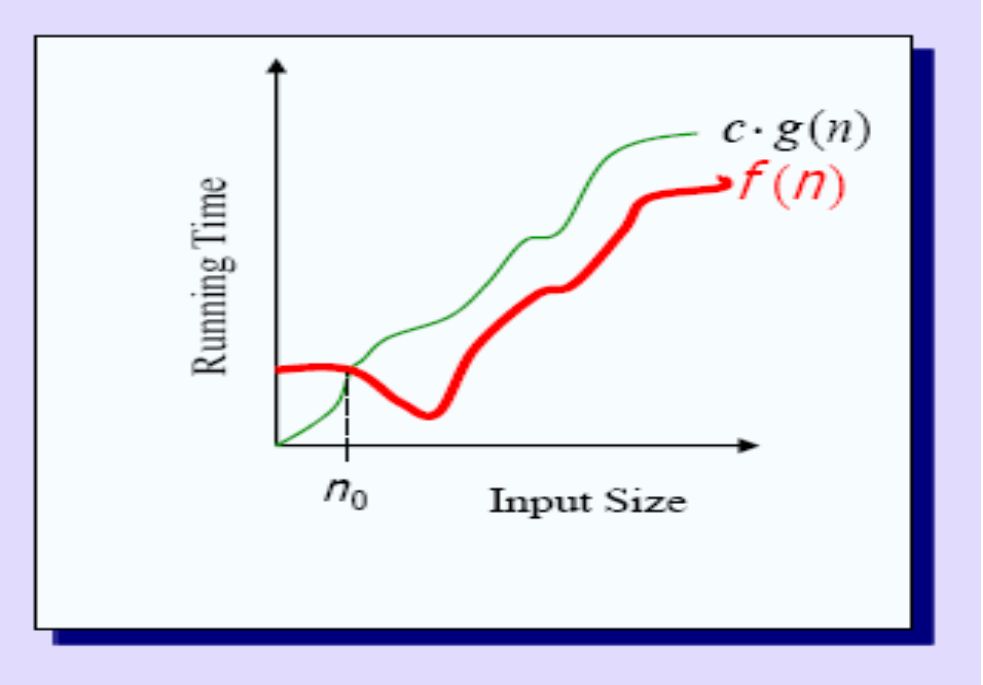
- We are mainly interested in worst-case bound
- Need to identify the worst-case scenario
- An algorithm that performs best in the worst-case scenario also performs best in the best case scenario (expectation)

Asymptotic notation

- The approach of counting primitive operations would be cumbersome to analyse complicated algorithms
- A simplified analysis that estimates the number primitive operation executed by an algorithm up to a constant factor by counting the steps of the algorithm
- Asymptotic notation facilitates analysis by getting rid of details

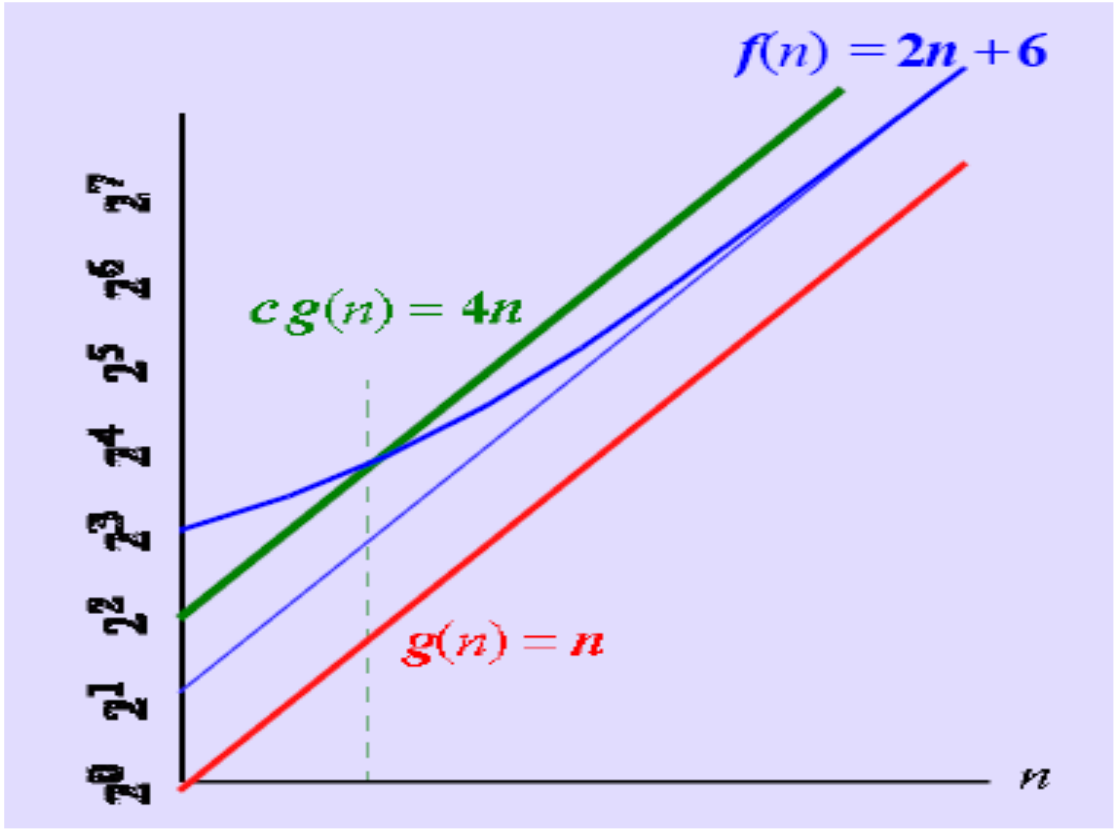
Asymptotic notation

- The “big-Oh” O-Notation
  - Provides asymptotic upper bound on the running time
  - f(n) is O(g(n)), if there exist constants “c” and “n_0” s.t. $f(n) \leq cg(n)$ for $n \geq n_0$
  - f(n) and g(n) are functions over nonnegative integers and non-decreasing functions



Asymptotic notation

- f(n) = 2n +6 and g(n) = n
- f(n) is O(g(n)), with c= 4 and n_0 = 3



Asymptotic notation

- How to find the order of a function?
  - If f(n) is a polynomial of degree “d”, then f(n) is $O(n^d)$
  - $\log n^x$ is O(log n) for any fixed $x > 0$
- Simple rule: drop lower order terms and constants

  Ex: 50 n log n is O(n log n), 7n – 2 is O(n), $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

  Note: Though 50 n log n is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

  Characterize the given function as closely as possible

Asymptotic analysis of running time

- Using O-notation, express the number of primitive operations executed as a function of input size
- How to compare asymptotic running times?
  - Algorithm that runs in O(n) time is better than that runs in $O(n^2)$
  - O(log n) is better than that O(n)
  - Hierarchy of running times: $\log n < n < n^2 < n^3 < a^n$

Example of asymptotic analysis

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X[j]
    A[i] ← a/(i+1)
return array A
```

1 step

Analysis: running time is $O(n^2)$

Example of asymptotic analysis

Algorithm prefixAverages2(X):

Input: An *n*-element array X of numbers.

Output: An *n*-element array A of numbers such that A[*i*] is the average of elements X[0], ... , X[*i*].

s ← 0

for i ← 0 **to** n-1 **do**

    s ← s + X[i]

    A[i] ← s/(i+1)

return array A

Analysis: Running time is O(n)

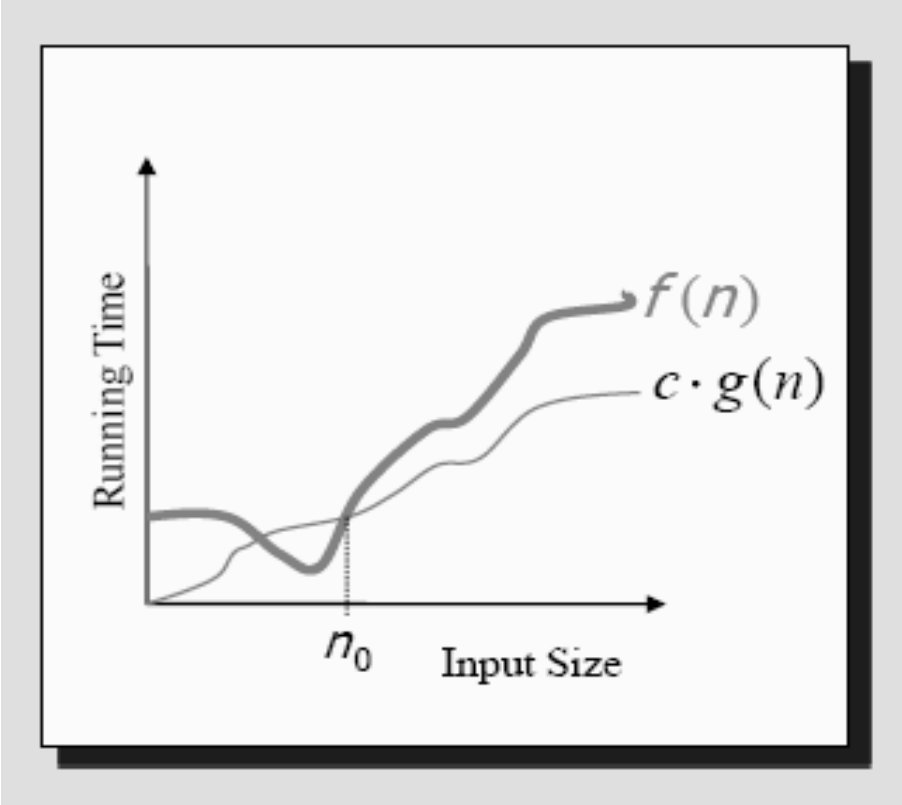
Classes of functions

- Logarithmic: $O(\log n)$
- Linear: $O(n)$
- Quadratic: $O(n^2)$
- Polynomial: $O(n^k)$ $(k \geq 1)$
- Exponential: $O(a^n)$ $(n > 1)$

Data Structures

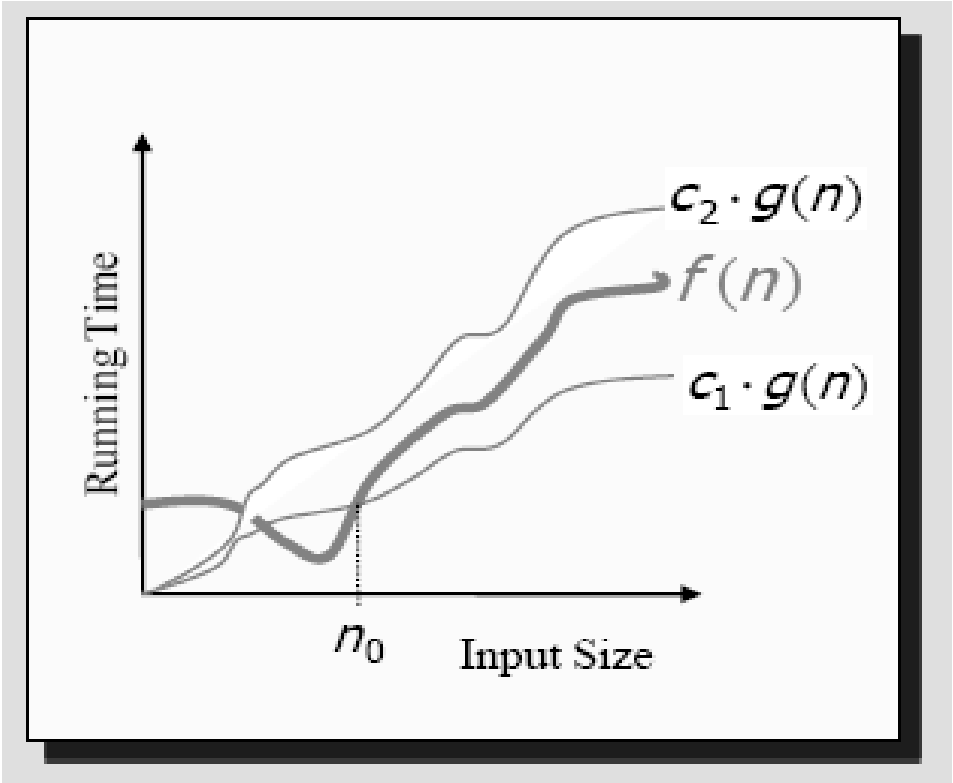
Asymptotic notation

- "Big-Omega" notation (Ω-notation)
  - Asymptotic lower bound on running time
  - f(n) is Ω(g(n)) if there exists constants "c > 0" and "$n_0 \geq 1$", s. t. c g(n) ≤ f(n), for all n ≥n_0.



Asymptotic notation

- "Big-Theta" notation (θ-notation)
  - Asymptotically tight bound
  - f(n) is θ(g(n)) if there exists constant "$c_1 > 0$", "$c_2 > 0$" and "$n_0 \geq 1$" s.t. $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$, for $n \geq n_0$



Asymptotic Notation

- Two more relatives of Big-Oh notation are:
  - “Little-Oh” notation: If f(n) is o(g(n)), then, for every c > 0, there should exist $n_0 > 0$, s.t. $f(n) \leq c\, g(n)$ for $n \geq n_0$
  - “Little-Omega” notation: If f(n) is ω(g(n)), then, for every c > 0, there should exist $n_0 > 0$, s.t. $c\, g(n) \leq f(n)$ for $n \geq n_0$

Importance of Asymptotics

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Stack operations: Running time

Algorithm push(o)

    if size() = N then
        indicate a stack-full error has occurred
    t ←t+1
    S[t] = o

Running time: O(1)

Algorithm pop()

    if(isEmpty()) then
        indicate a stack-empty error has occurred
    e ← S[t] {e is a temporary variable}
    S[t] ← null
    t ← t-1
    return e

Running time: O(1)

Queue Operations: Running time

Algorithm enqueue(e)

    if (size() == N-1)

        raise QueueFull exception

    Q[r] = e

    r ← (r+1) mod N

Running time: O(1)

Algorithm dequeue()

    if isEmpty() then

        raise QueueEmpty Exception

    temp ← Q[f]

    Q[f] = null

    f ← (f+1) mod N

    return temp

Running time: O(1)

List Operations: Running time

Algorithm insert(e, p)

    for i=n-1, n-2, . . ., p do

        S[i+1] ← S[i] {Make room for “e” to be inserted}

    S[p] ← e

    n ← n+1

Running time: O(n)

Algorithm removeAtPosition(p)

    e ← S[p] {e is a temporary variable}

    for i=p, p+1, . . ., n-2 do

        S[i] = S[i+1] {fill in for the removed element}

    n ←n-1

Running time: O(n)

Linked list: Running time

- Assume that a list is implemented using a singly linked list
- The list is maintained in sorted order
- We have reference to the head of the list
- The running time of insert(e) operation?
- The running time of delete(e) operation?

Sorting

- Storing data in an ordered (increasing or decreasing or alphabetical, etc.) manner
- Searching becomes efficient when data is maintained sorted order
- Computer graphics and computational geometry
- A large number of sorting algorithms representing different design techniques have been developed

Sorting Problem

Input: a sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A permutation or reordeing $(a_1', a_2', \ldots, a_n')$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Requirements for the output: output should be a permutation of the input

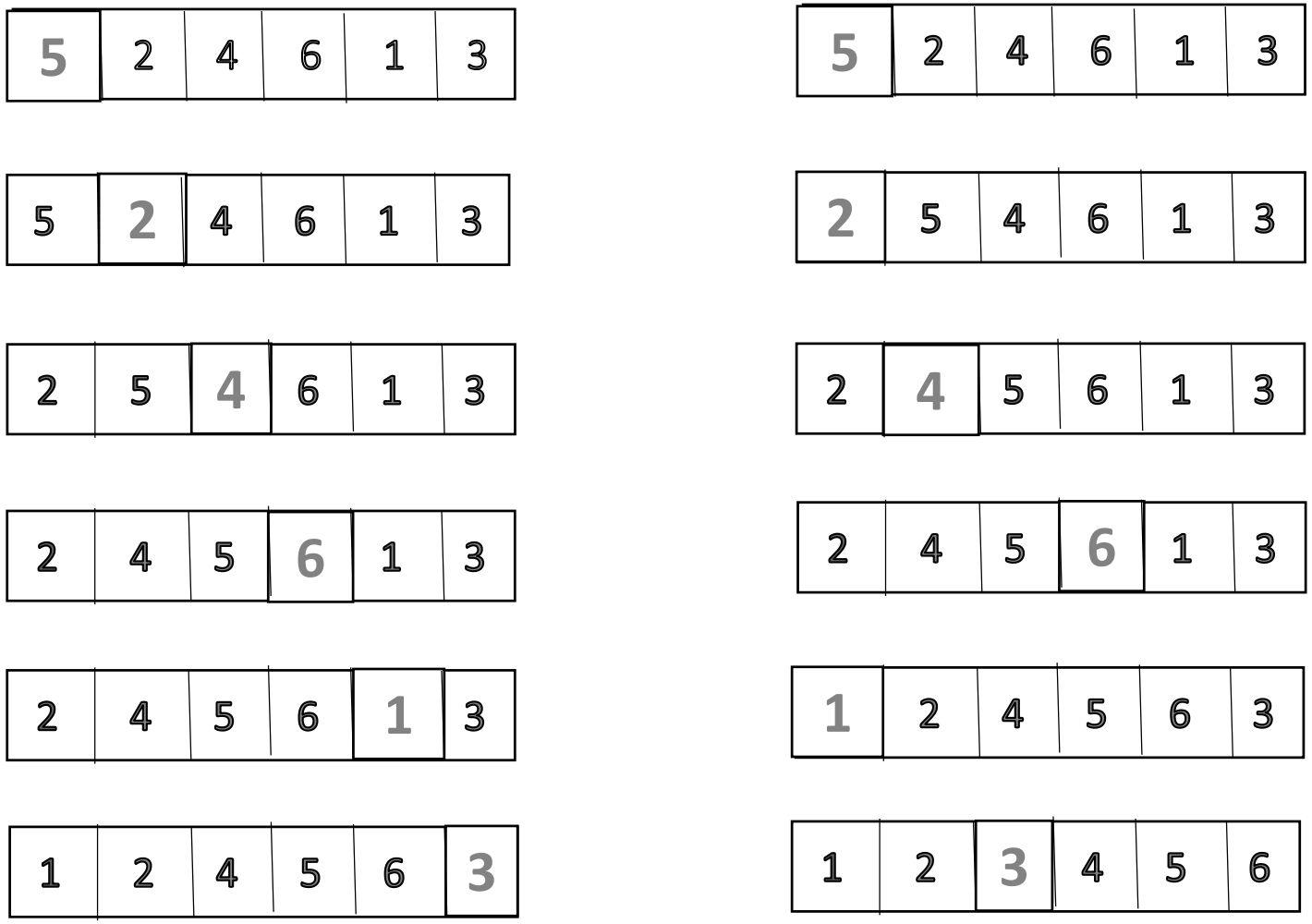
Factors affecting the running time: The input size, how sorted the input sequence is, and the algorithm used

Insertion Sort

- At any point of time, the given sequence can be divided into two parts:
  - Sorted part
  - Unsorted part
- Initially, the sorted part is empty
- Pick the first element from the unsorted part
- Place the picked element in the correct position in the sorted part
- Repeat the same process with the remaining elements

Insertion Sort

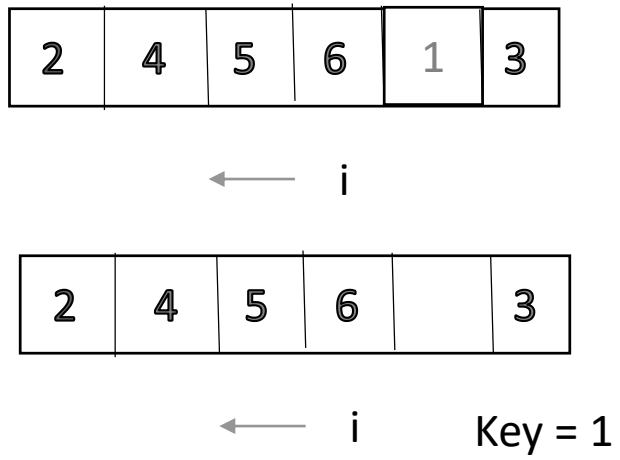
- Strategy



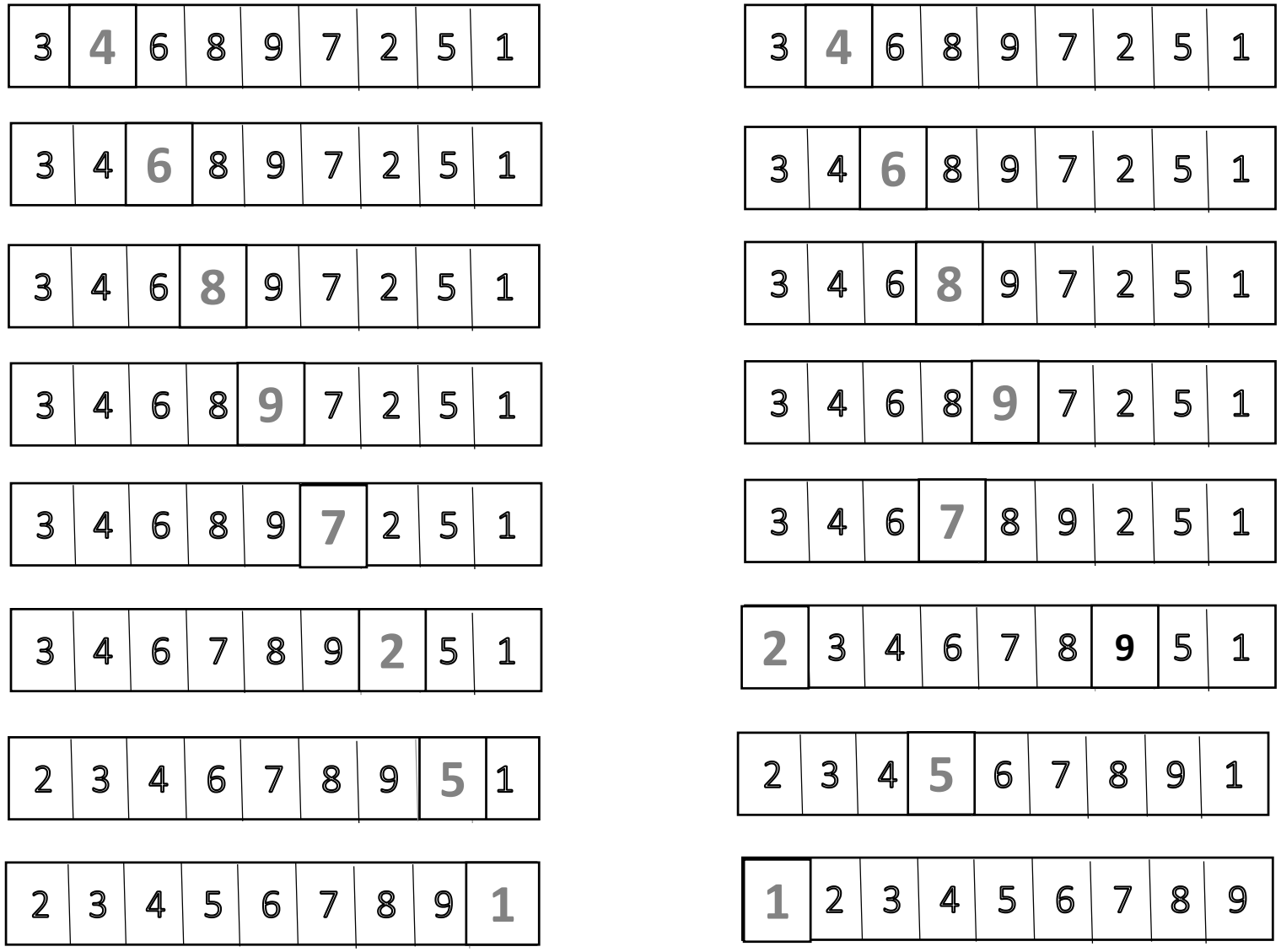
Insertion sort algorithm

Algorithm Insertion_Sort(A)
Input: A[0. .n-1] – an array of integers
Output: a permutation of A such that A[0] ≤ A[1] ≤ . . . ≤ A[n-1]

```
for j ← 1 to n-1 do
        key ← A[j]
        {insert A[j] into the sorted
                sequence A[0. .j-1]}
        i ← j-1
        while i ≥ 0 and A[i] > key do
                A[i+1] ← A[i]
                i - -
        A[i+1] ← key
```



Insertion Sort: Example



Analysis of insertion sort

for j ← 1 to n-1 do	n
key ← A[j]	n-1
{insert A[j] into the sorted sequence A[0. .j-1]}	
i ← j-1	n-1
while i ≥ 0 and A[i] > key do	$\sum_{j=1}^{n-1} t_j$
A[i+1] ← A[i]	$\sum_{j=1}^{n-1} (t_j - 1)$
i - -	$\sum_{j=1}^{n-1} (tj - 1)$
A[i+1] ← key	n-1

Analysis of insertion sort

- Best-case: $O(n)$
- Worst-case: $O(n^2)$
- Average case: $O(n^2)$

Data Structures

Divide and Conquer

- A technique to solve a computational problem by dividing it into one or more subproblems, recursively solve the subproblems, and then merge the solutions to the subproblems
  - **Divide:** The problem is divided into a number of subproblems which are smaller instances of the original problem
  - **Conquer:** The subproblems are solved recursively
  - **Combine:** Combine the solutions to the subproblems to get the solution to the original problem
- "n" is the size of the problem, "S(n)" is the problem to be solved
- S(n) is divided into $S(n_1)$, $S(n_2)$, . . ., $S(n_k)$, where $n_i < n$ for i = 1,2, . . ., k
- Solve $S(n_1)$, $S(n_2)$, . . ., $S(n_k)$
- Combine the solutions of $S(n_1)$, $S(n_2)$, . . ., $S(n_k)$ to get the solution of S(n)

Merge Sort

- Given with an array S of elements/keys
- **Divide:** If S has zero or one element, return S directly. Otherwise (that is, if S has at least two elements), remove all elements from S and put them in two sequences, S_1 and S_2, each containing half of the elements of S (that is, S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$)
- **Conquer:** Sort sequences S_1 and S_2 using Merge Sort
- **Combine:** Merge the sorted sequences S_1 and S_2 into one sorted sequence and put it in S

Merge Sort Algorithm

```
Algorithm Merge_Sort(A, l, r)
        If(l < r)
                center = (l+r)/2
                Merge_Sort(A, l, center)
                Merge_Sort(A, center + 1, r)
                Merge(A, l, center, r)
```

Merge algorithm

Algorithm Merge(A, l ,center, r)

$k \leftarrow l$, $n_1 \leftarrow$ center-l+1, $n_2 \leftarrow$ r-center

$S_1 \leftarrow A[l \ldots center]$

$S_2 \leftarrow A[center+1 \ldots r]$

$i \leftarrow 0$, $j \leftarrow 0$

while ($i < n_1$ and $j < n_2$) do

if $S_1[i] \leq S_2[j]$ then

$A[k] = S_1[i]$

$i \leftarrow i+1$

$k \leftarrow k+1$

else

$A[k] = S_2[j]$

$j \leftarrow j+1$

$k \leftarrow k+1$

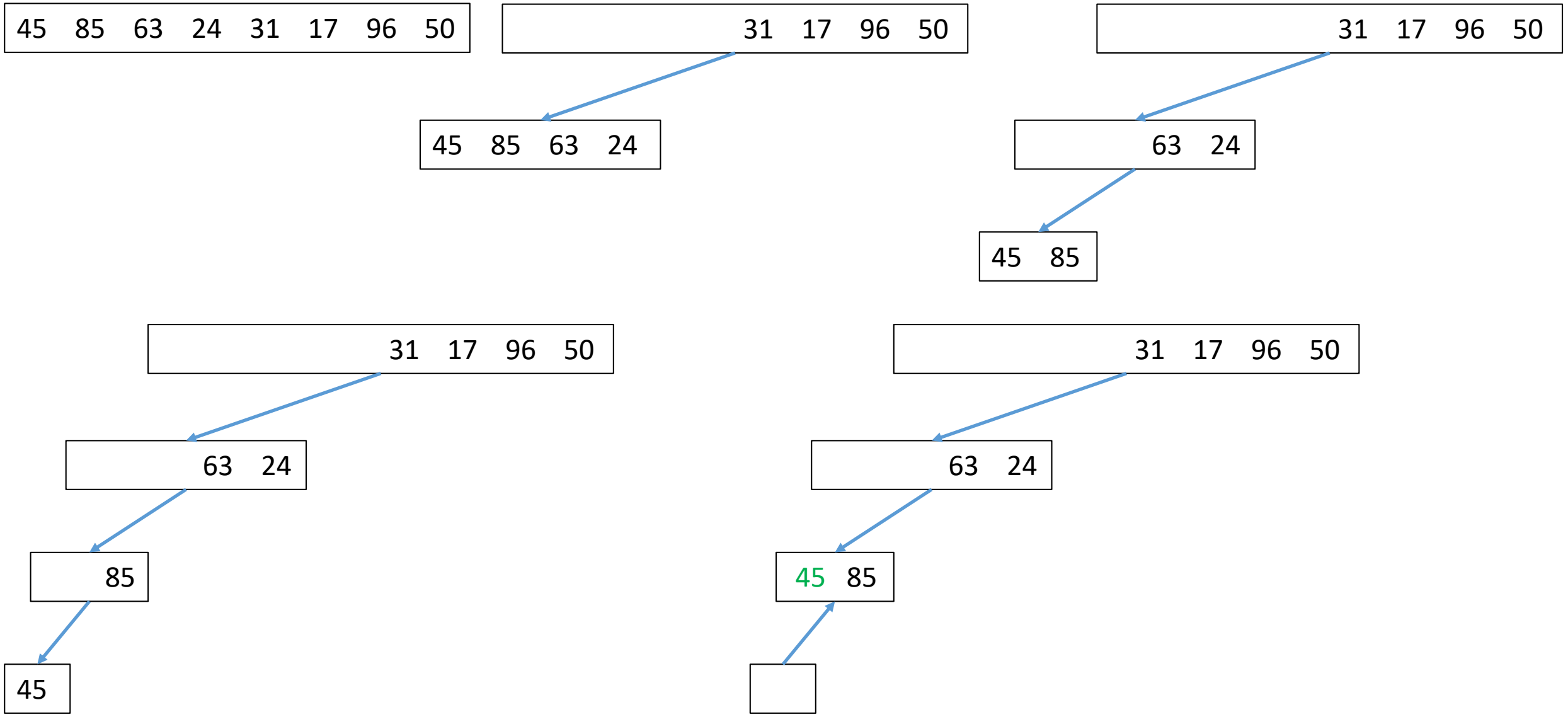
if (i < n1)

Copy the remaining elements from S_1 to A

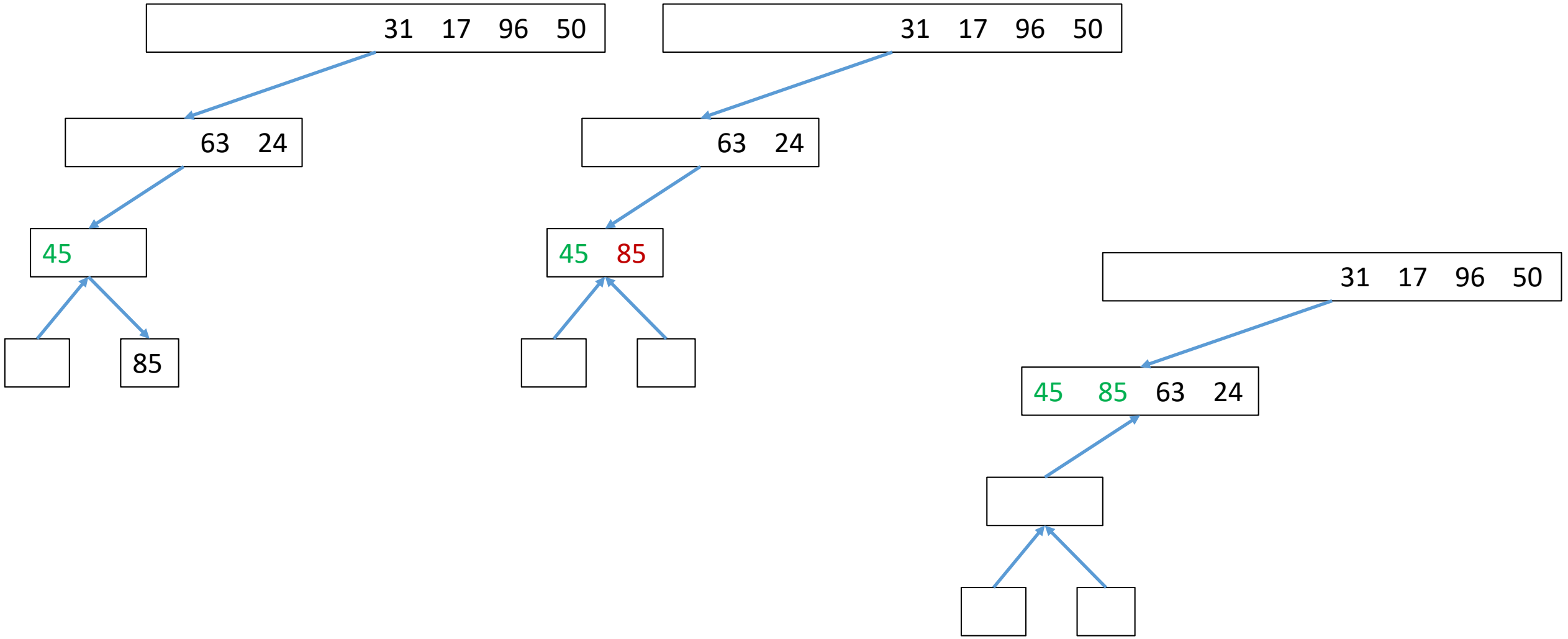
if(j < n2)

Copy the remaining elements from S_2 to A

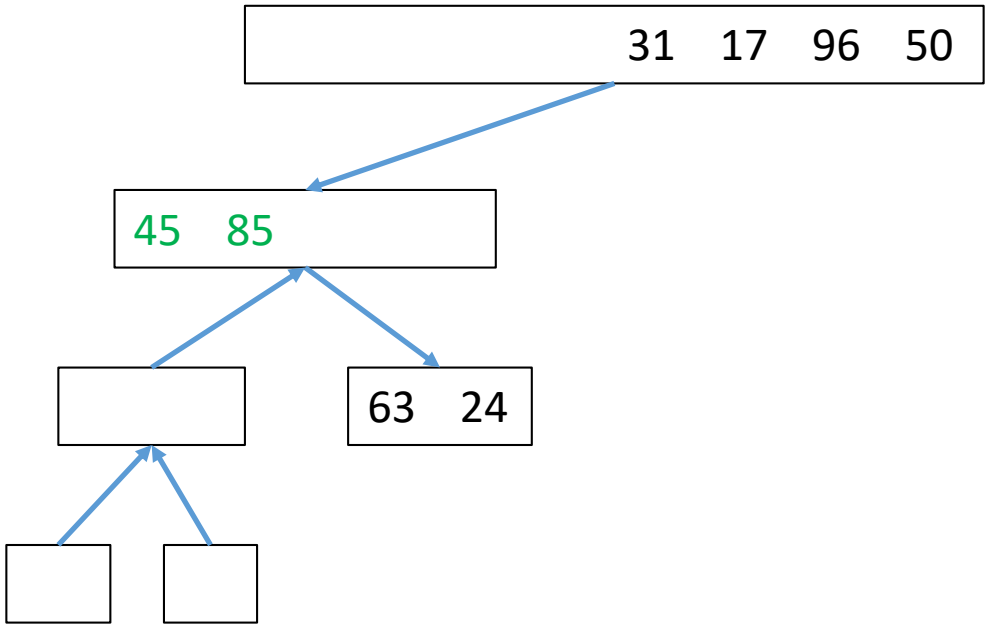
Merge Sort: Example

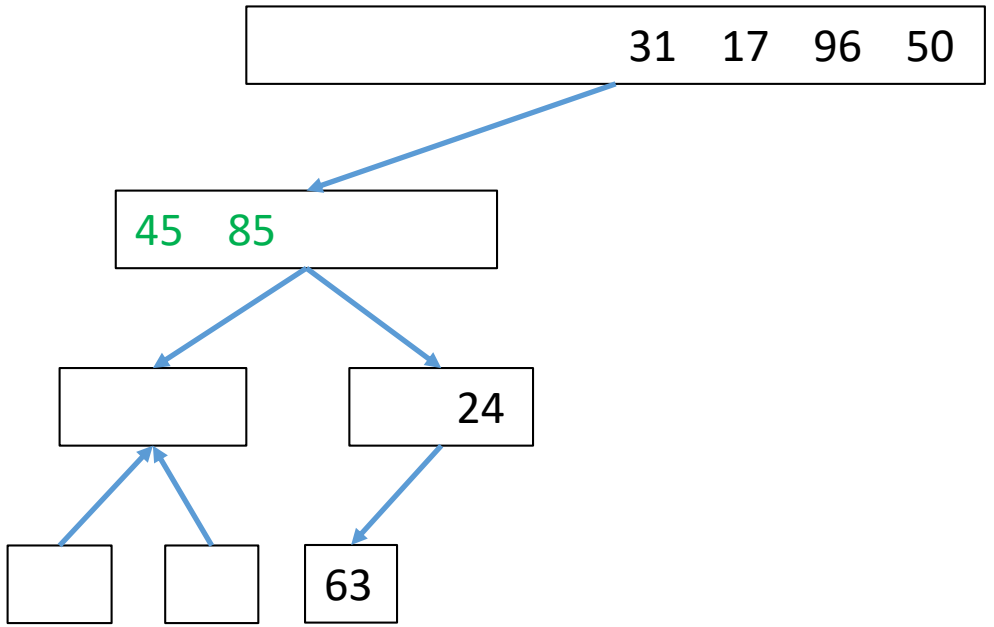


Merge Sort: Example

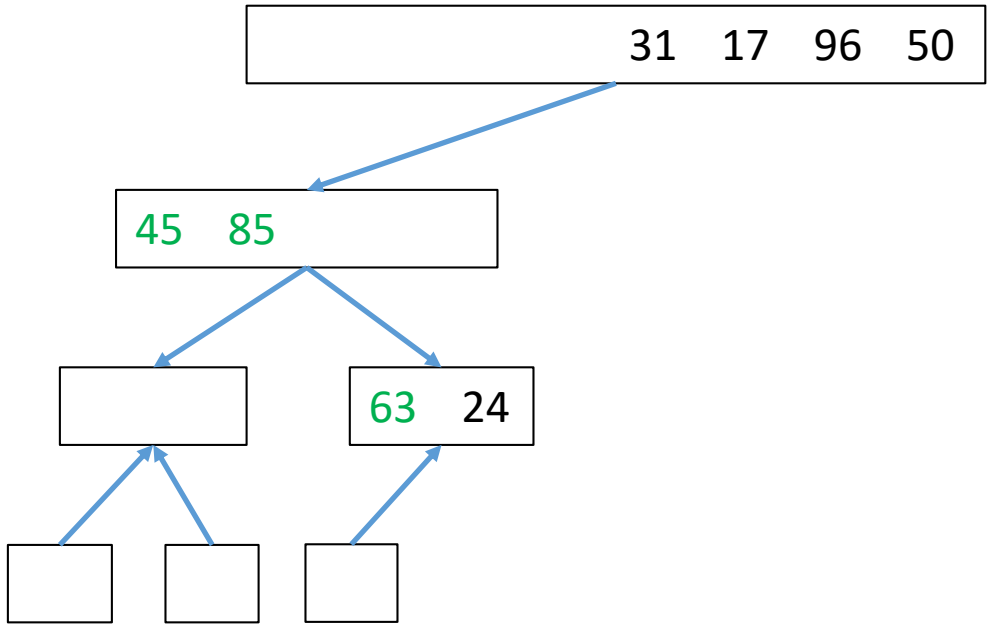


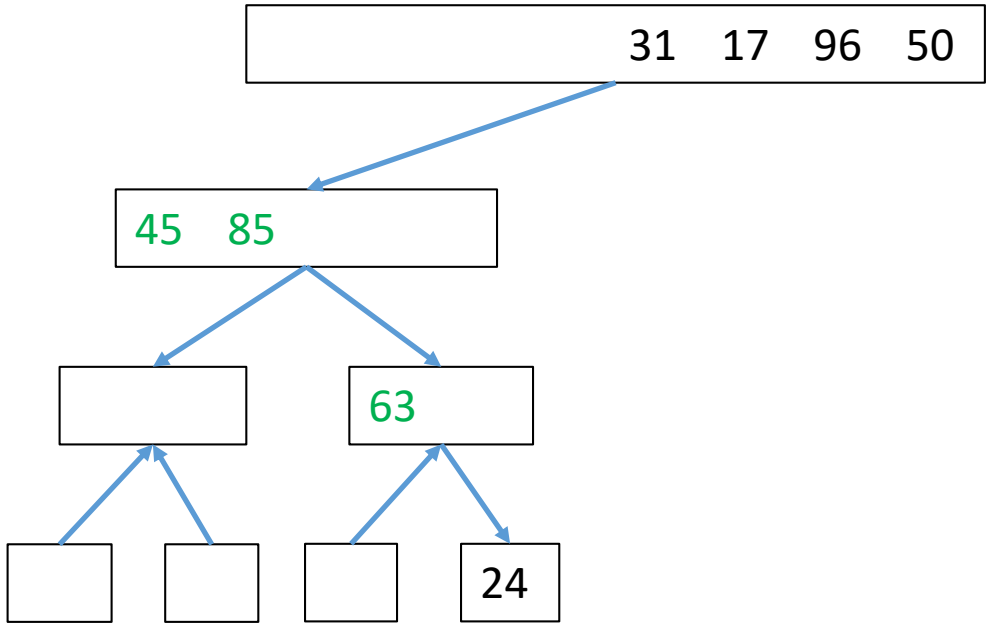
Merge Sort: Example



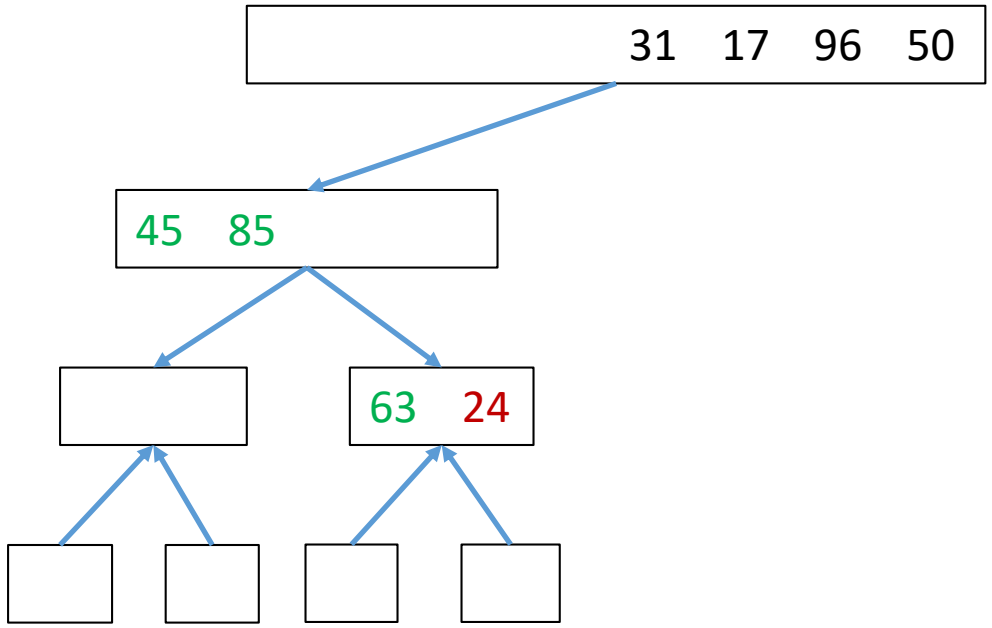


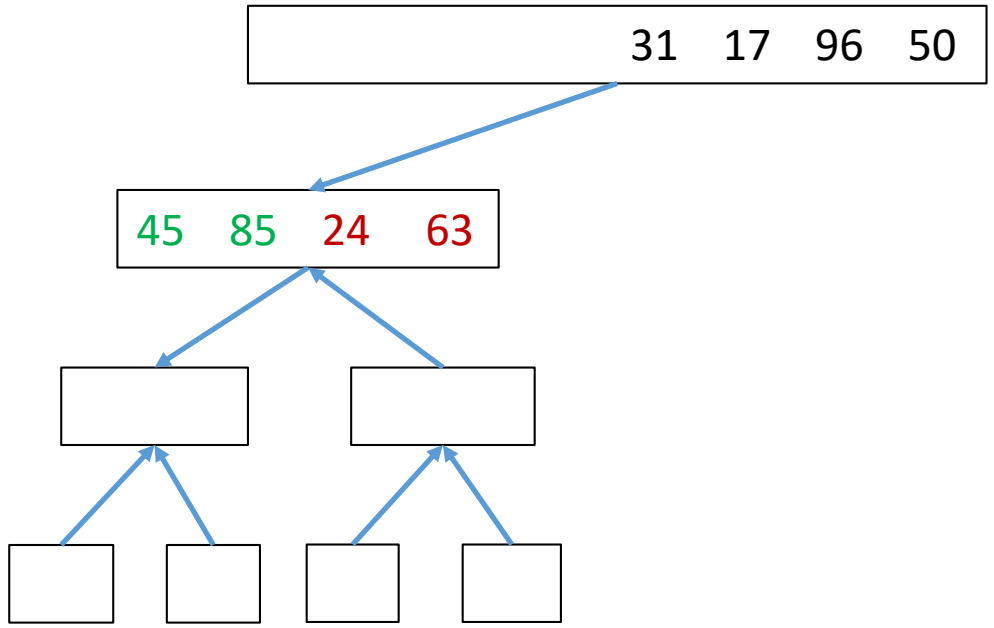
Merge Sort: Example



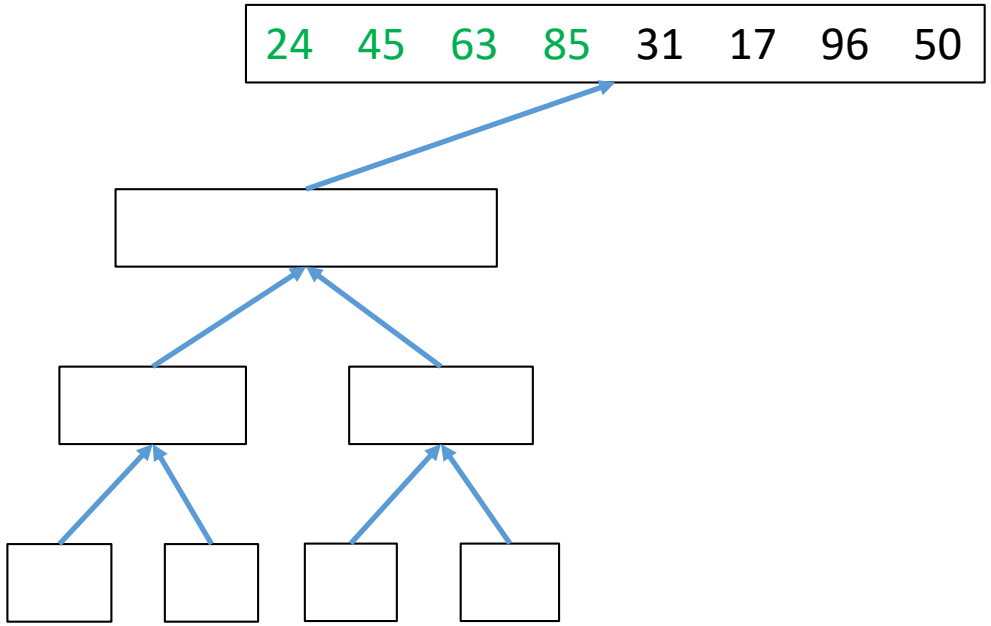


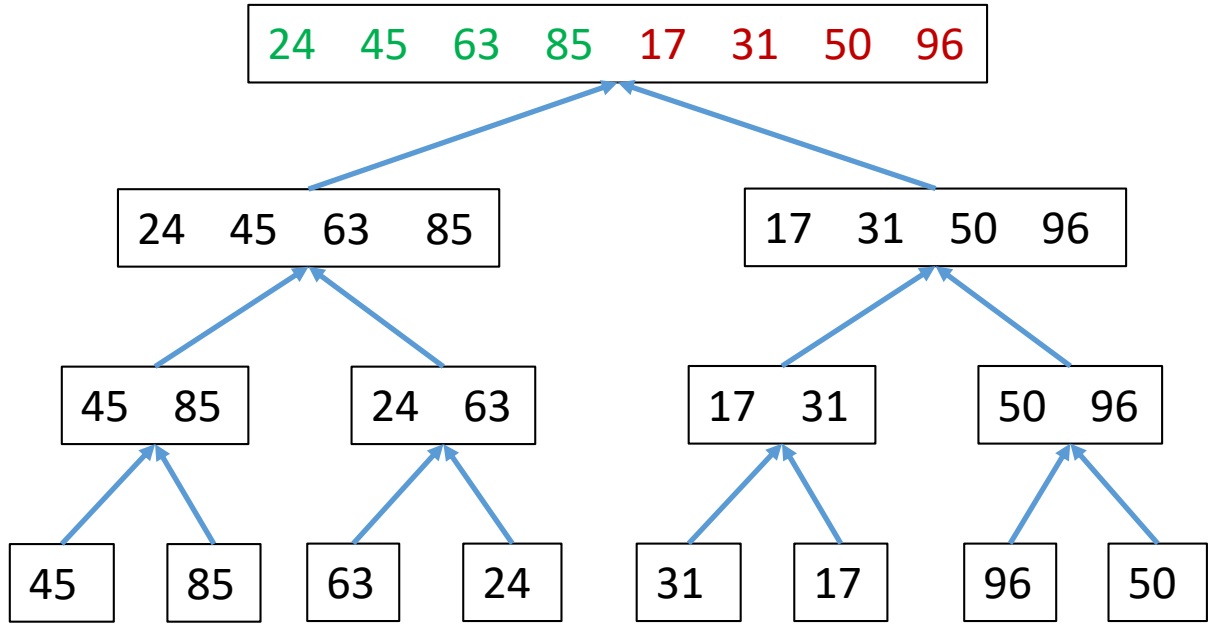
Merge Sort: Example



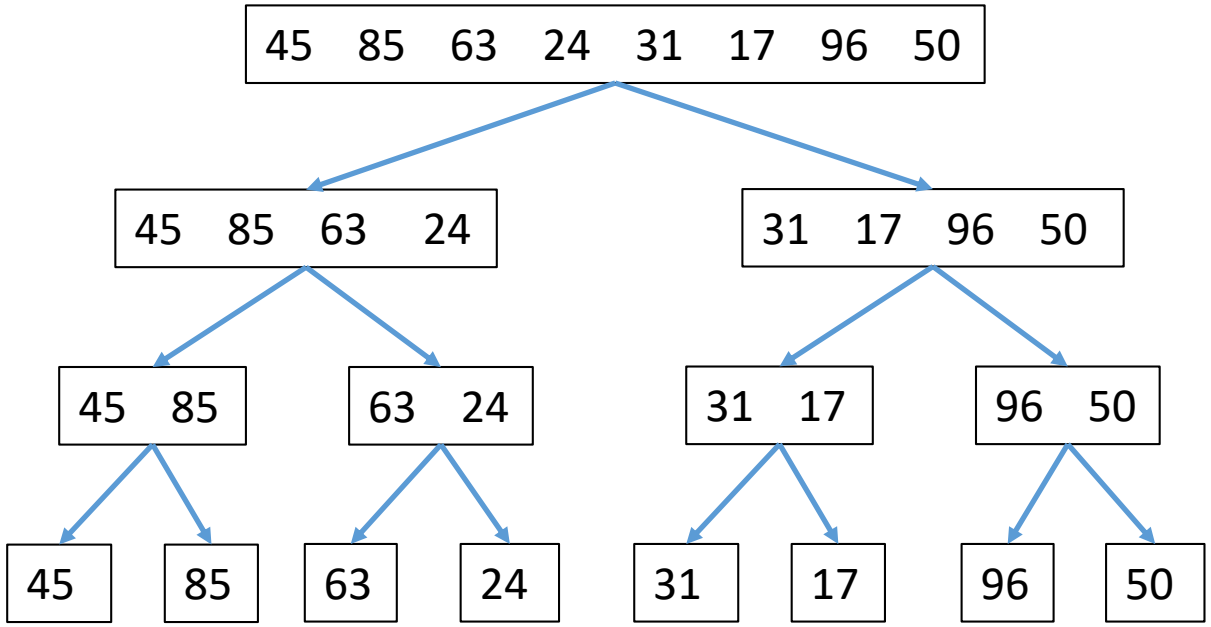


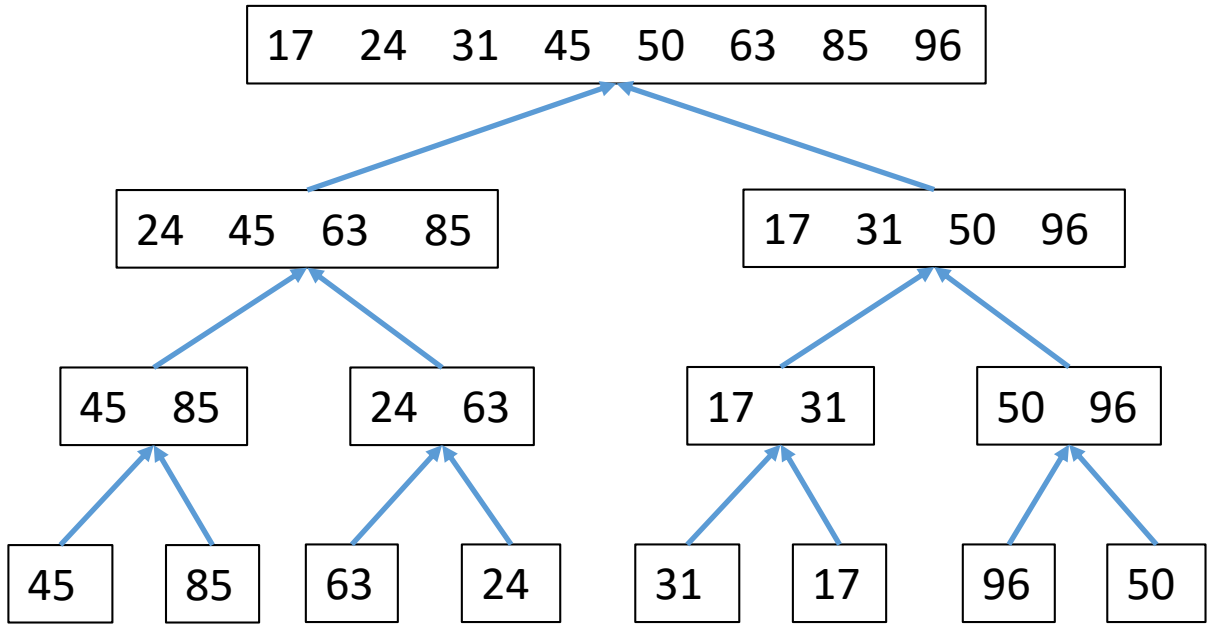
Merge Sort: Example



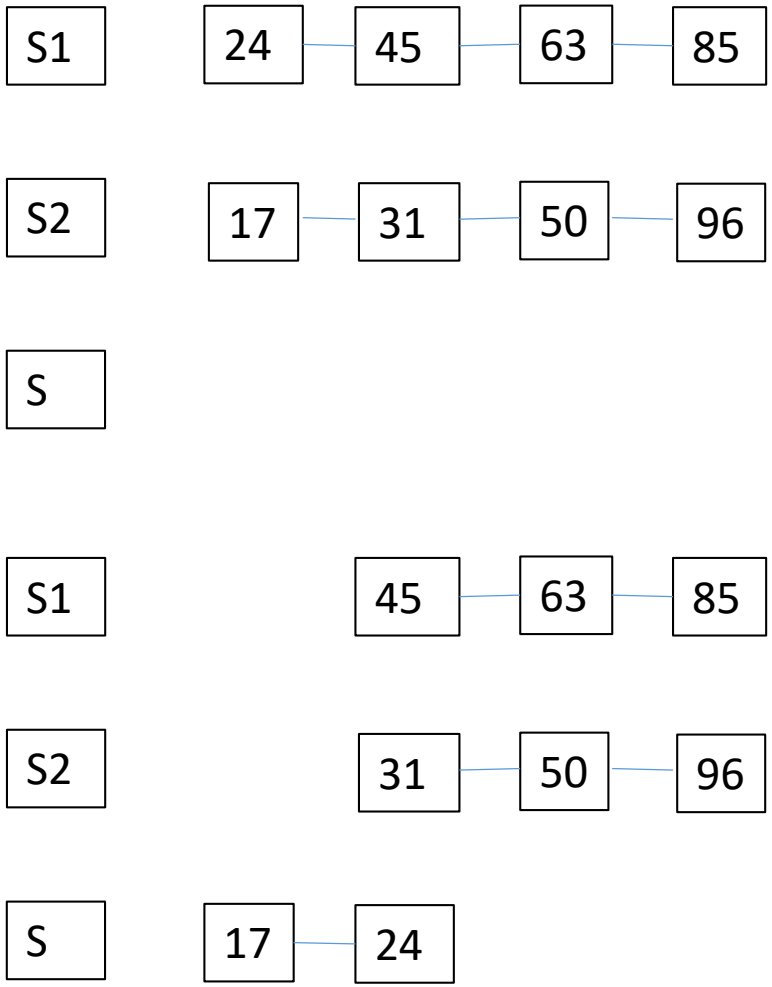


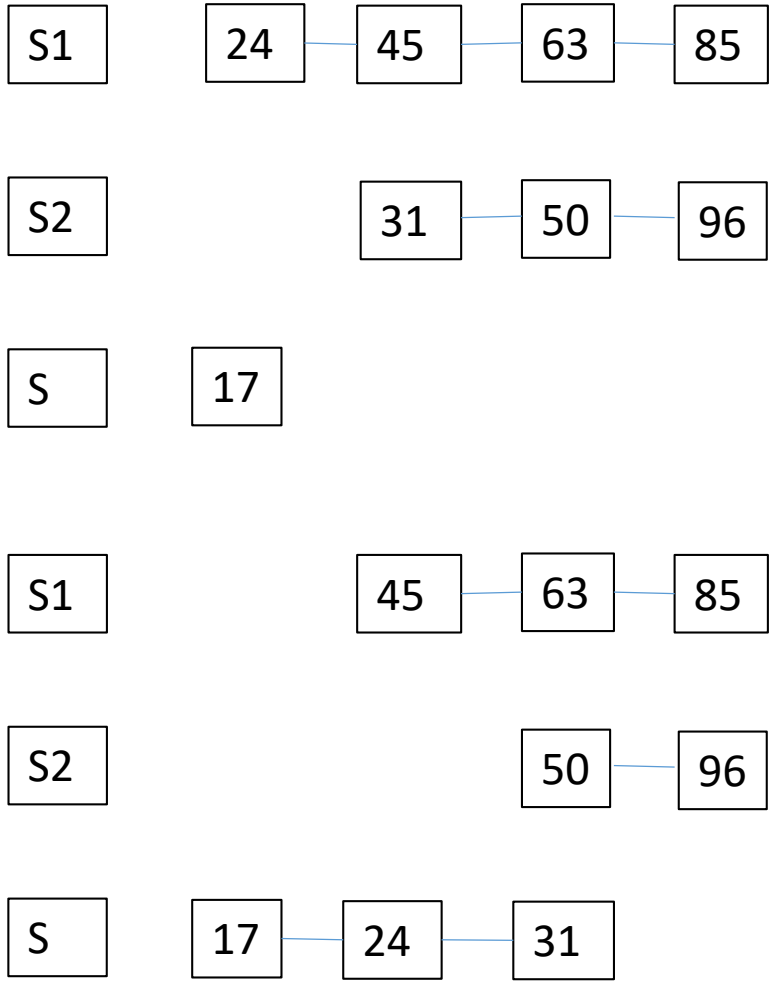
Merge Sort: Example



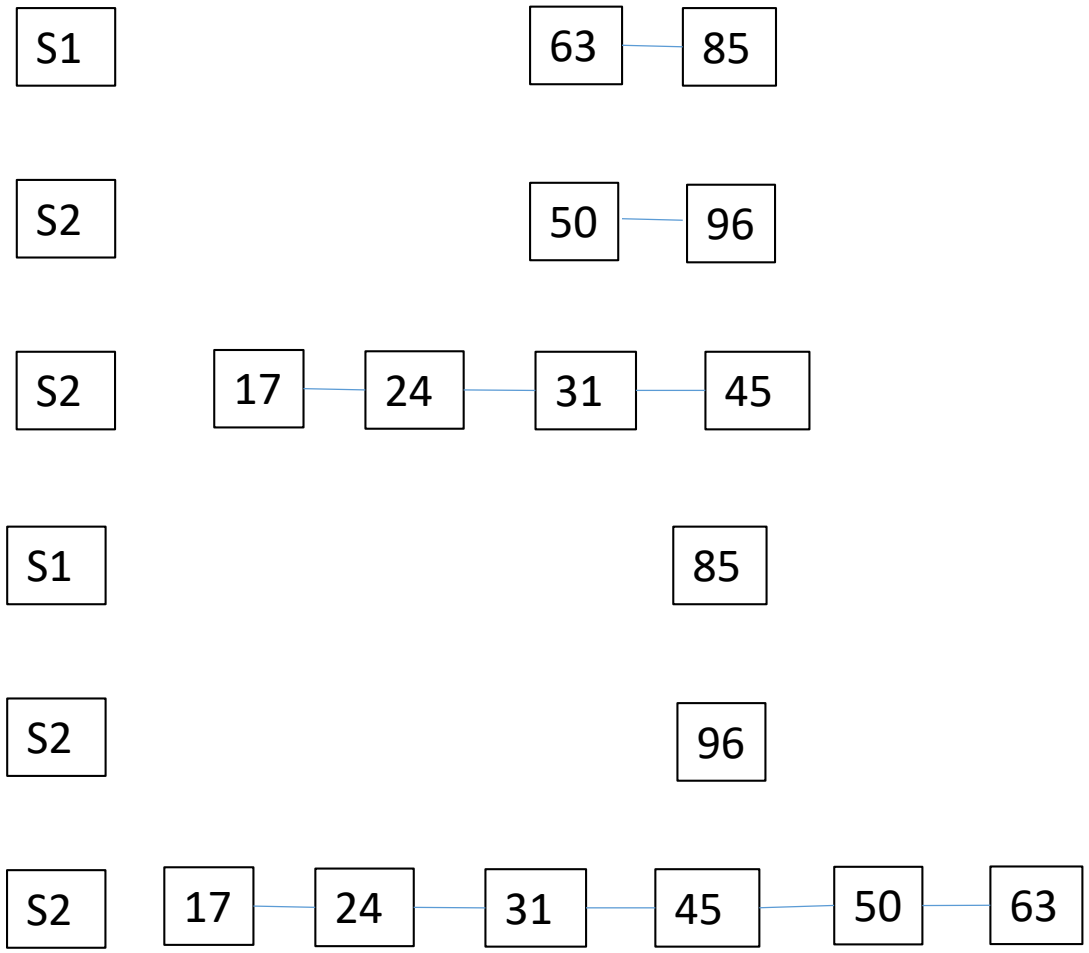


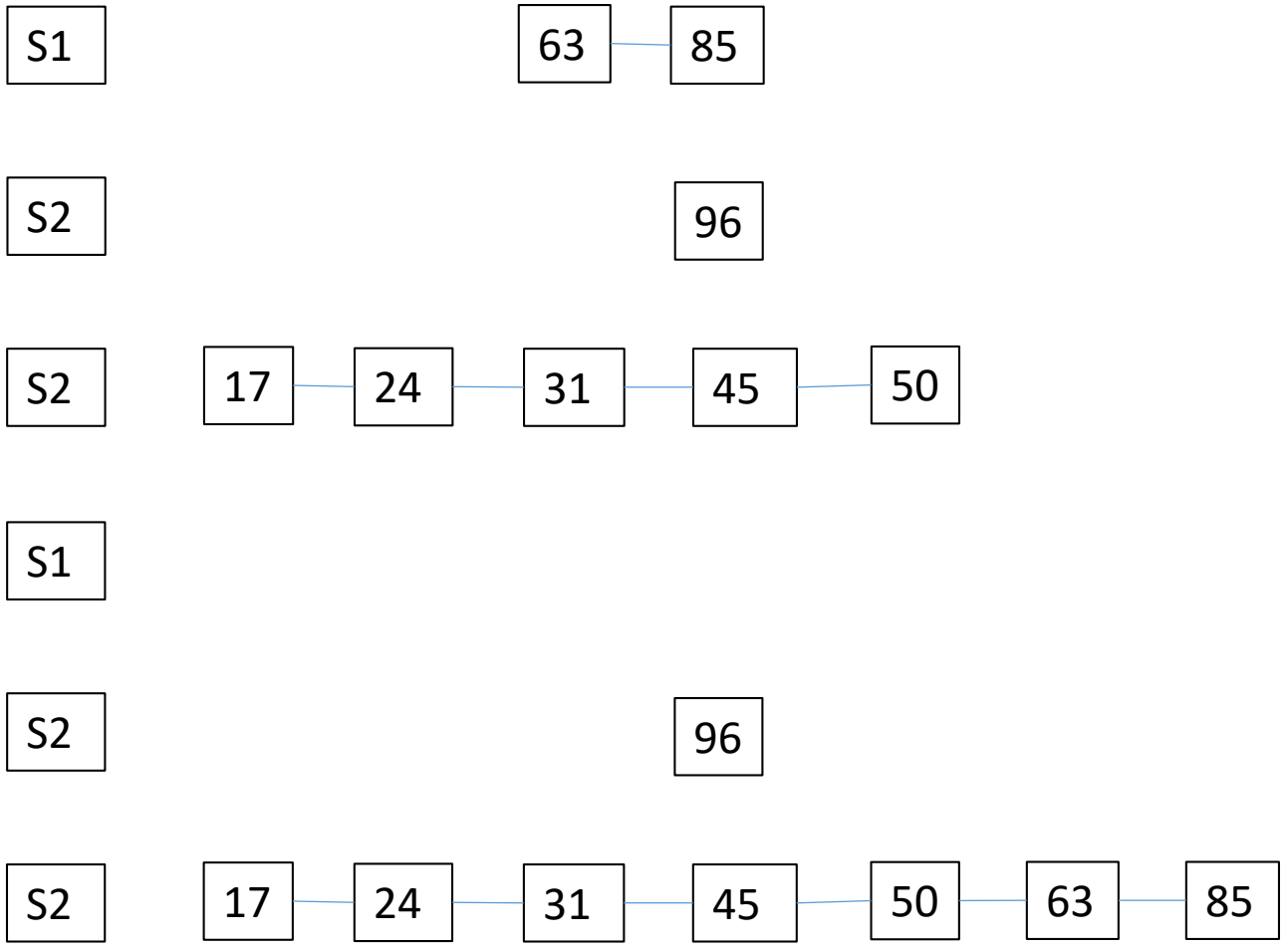
Merge Sequences



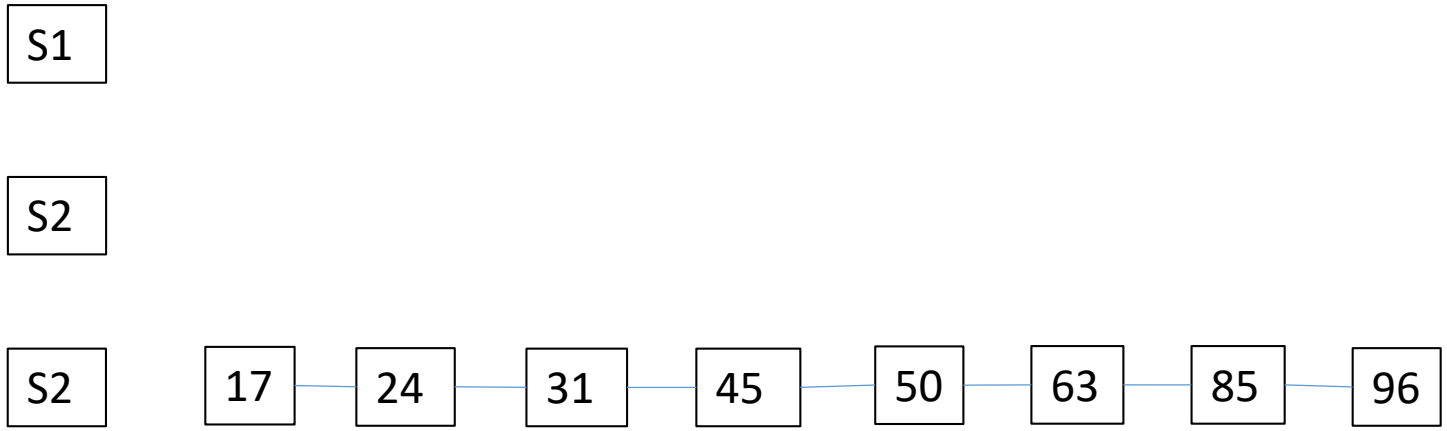


Merge Sequences





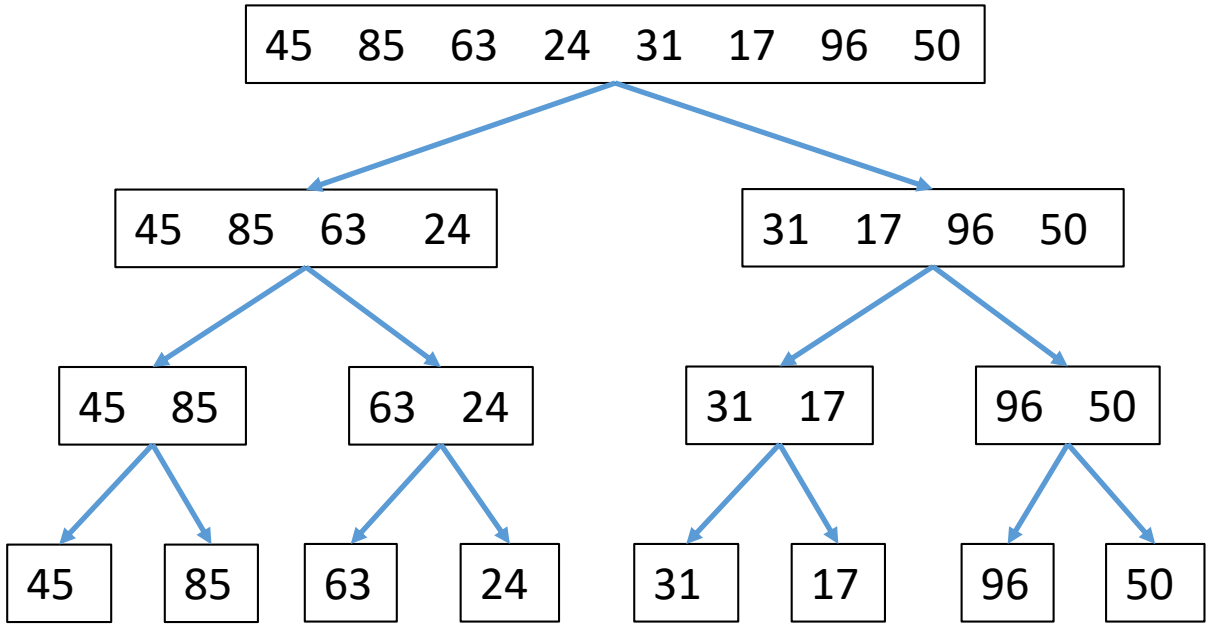
Merge Sequences

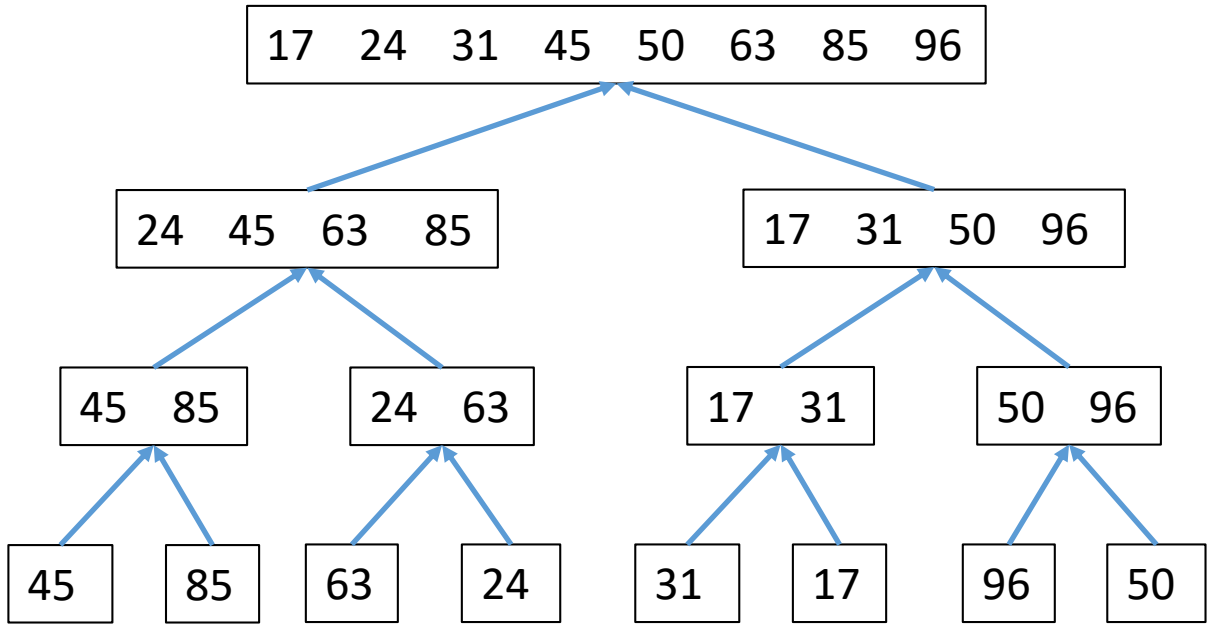


Analysis of Merge method

- The merge algorithm is of order O(n1 + n2), where n1 and n2 are sizes of sequences S1 and S2, respectively
- If n1 and n2 are sizes of sequences S1 and S2, respectively, then the number of comparisons in the worst case?
- The number of comparisons in the best case?

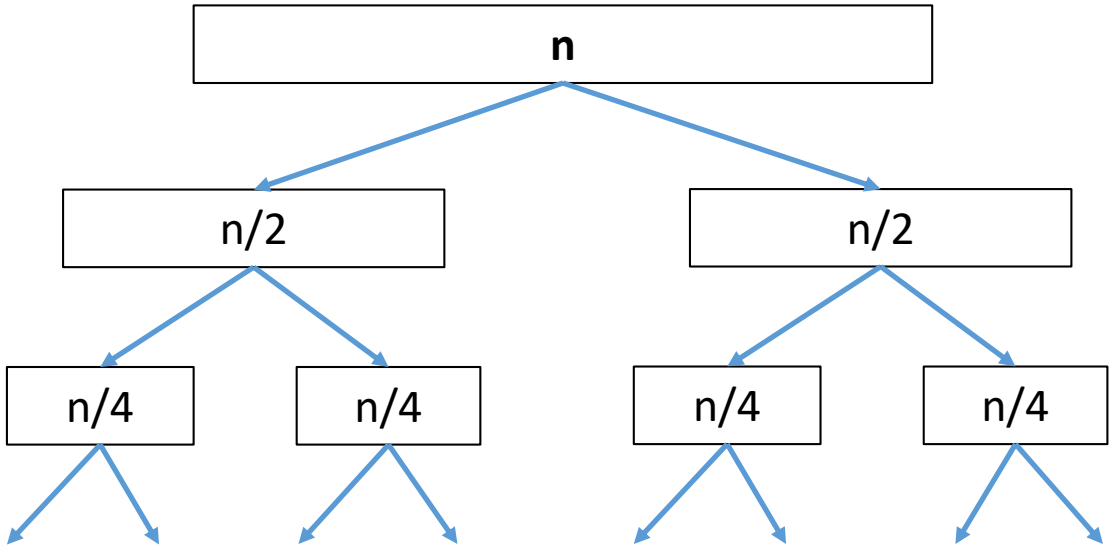
Analysis of Merge Sort





Analysis of Merge Sort

- The size of input, n, is a power of 2
- The time spent at node v:
  - Time spent in division
  - Time spent in merge
- The time spent at node v which is at depth "i" is $O(n/2^i)$
- The time spent at a depth is: $O(2^i\ n/2^i)$
- The height tree is log(n)
- Running time is: O(n log n)
- We log n to represent $\log_2 n$



Data Structures

Analysis of divide-and-conquer algorithms

- If an algorithm contains recursive calls to itself, then its running time can be expressed by a recurrence equation or recurrence
- Running time of algorithm on input size n is expressed in terms of running time on smaller input sizes
- Assume that, if $n \leq c$, then the problem can be solved directly
- If $n > c$, then the problem is divided into "a" subproblems of size "n/b"
- D(n) is the time to divide and C(n) is the time to combine

$$T(n) = \begin{cases} \text{time to solve the trivial problem} & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of divide-and-conquer algorithms

- Assume that n is a power of 2
- Recurrence for merge sort:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Solving recurrences

- Four methods to solve recurrences
- Iterative substitution method
  - Expand the recurrence by substitution and observe patterns
- The recursion tree method
  - Similar to iterative substitution method, visual approach
- Substitution method (guess-and-test method)
  - Make an educated guess of the closed form solution and then justify the solution usually by induction
- The master method
  - A general and cook-book method to determine asymptotic characterization of a wide variety of recurrences

Iterative substitution method

- Substitute the general form of the recurrence for each occurrence of function T on the right hand-side
- Substitutions are done with the hope that at some point we see a pattern that can converted into a general closed form equation (with T appearing only on the left hand-side

Iterative substitution method

- Ex: Consider the recurrence equation of merge-sort

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

$$2T(n/2) = 2(2(T(n/4)) + n/2)$$
$$= 4T(n/4) + n$$
$$T(n) = 4T(n/4) + 2n$$

$$4T(n/4) = 4(2(T(n/8)) + n/4)$$
$$= 8T(n/8) + n$$
$$T(n) = 8T(n/8) + 3n$$

Iterative substitution method

Continuing in this manner, we obtain

$$T(n) = 2^k T(n/2^k) + kn$$

Using k = log n (log n is $\log_2 n$), we obtain

$$T(n) = n\,T(1) + n \log n$$

$$= n + n \log n$$

The recursive tree method

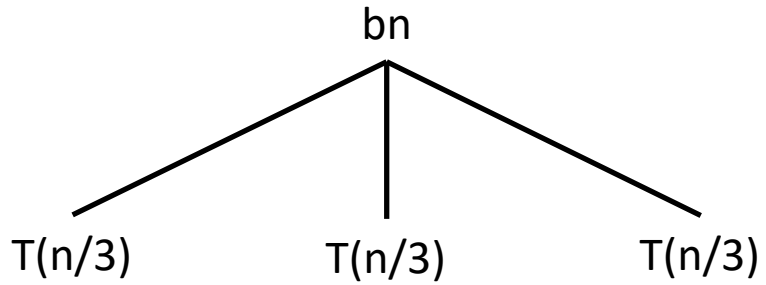
- This technique also uses repeated substitutions, not in an algebraic manner, but in visual manner
- Draw a tree R where each node represents a different substitution of the recurrence equation
- Each node v of R has a value of the argument n of T(n)
- An overhead is associated with each node v in R
- Solution: the summation of overheads associated with all nodes in R

The recursive tree method

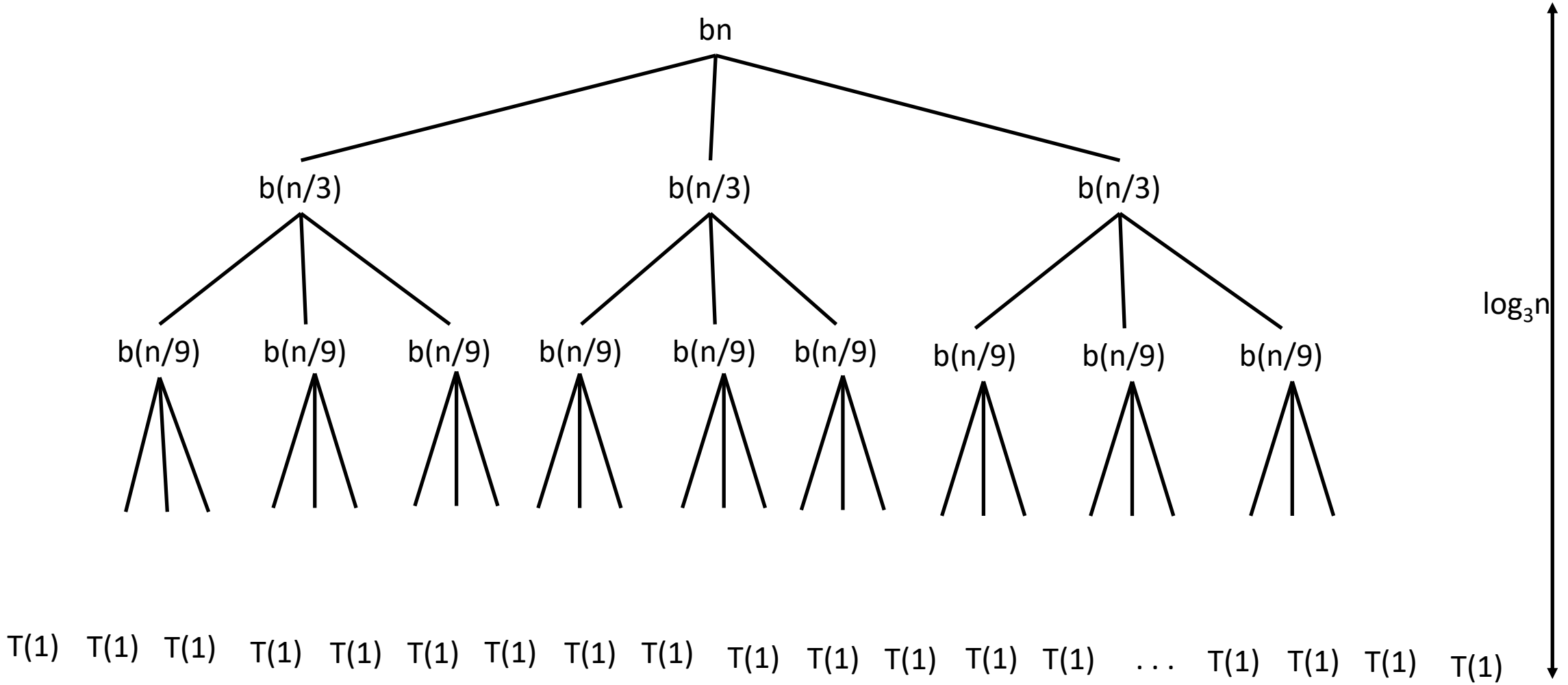
- Ex: Consider the below given recurrence (n is an exact power of 3)

$$T(n) = \begin{cases} c & \text{if } n < 3, \\ 3T(n/3) + bn & \text{otherwise} \end{cases}$$

What does this recurrence equation represent?



The recursive tree method



The guess-and-test method

- First make an educated guess as to what a closed form solution of the recurrence equation might look like
- Then justify the guess usually by induction
- Powerful yet must be able to guess the form of the solution

The guess-and-test method

- Ex: Consider the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n < 2, \\ 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

- First guess: $T(n) \leq cn \log n$, for some constant $c > 0$
- Assume that this first guess is inductive hypothesis for input sizes smaller than n
- This guess holds true for m<n, in particular m = $\lfloor n/2 \rfloor$
- $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log (\lfloor n/2 \rfloor)$

The guess-and-test method

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2c\lfloor n/2 \rfloor \log (\lfloor n/2 \rfloor) + n \\
&\leq cn \log (n/2) + n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&\leq cn \log n
\end{aligned}
$$

The last step holds as long as $c \geq 1$

The guess-and-test method

- We have to show that our guess $T(n) \leq cn \log n$ works for boundary conditions as well
- According to the guess, $T(1) \leq c\ 1 \log 1 = 0$
- According asymptotic notation, we have to show that $T(n) \leq cn \log n$, for $n \geq n_0$
- Do not consider the problematic boundary condition in the induction proof

$$T(n) = \begin{cases} 1 & \text{if } n < 2, \\ 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

- T(4), T(5), . . . do not directly depend on T(1)
- Only T(2) and T(3) depend on T(1)

The guess-and-test method

- Replace T(1) with T(2) and T(3) (base cases of the induction proof)
- $n_0 = 2$
- T(2) = 4 and T(3) = 5
- Now, we can complete the induction proof by choosing c s.t. T(2) ≤ c 2 log 2 and T(3) ≤ c 3 log 3

The master method

- It is a cook-book based approach for determining asymptotic characterization
- Used for recurrences of the form:

$$T(n) = \begin{cases} c & \text{if } n < d, \\ aT(\lfloor n/b \rfloor) + f(n) & \text{otherwise} \end{cases}$$

where d ≥ 1 is an integer constant and a > 0, c > 0, and b > 1 are real constants and f(n) is a function that is positive for n ≥ d

The master method

- Assume that T(n) and f(n) be as defined previously
- The master theorem
  - If there is a small constant ε > 0, s. t. f(n) is O($n^{log_b a - \varepsilon}$), then T(n) is θ($n^{log_b a}$)
  - If there is a small constant k ≥ 0, s. t. f(n) is θ($n^{log_b a}\ log^k n$), then T(n) is θ($n^{log_b a}\ log^{k+1} n$)
  - If there are small constants ε > 0 and δ < 1, s. t. f(n) is Ω($n^{log_b a + \varepsilon}$) and af(n/b) ≤ δf(n), for n ≥ d, then T(n) is θ($f(n)$)

Data Structures

The master method

- Assume that T(n) and f(n) be as defined previously
- The master theorem
  - If there is a small constant ε > 0, s. t. f(n) is O($n^{log_b a - \varepsilon}$), then T(n) is θ($n^{log_b a}$)
  - If there is a small constant k ≥ 0, s. t. f(n) is θ($n^{log_b a}\ log^k n$), then T(n) is θ($n^{log_b a}\ log^{k+1} n$) (log n is $\log_2 n$)
  - If there are small constants ε > 0 and δ < 1, s. t. f(n) is Ω($n^{log_b a + \varepsilon}$) and af(n/b) ≤ δf(n), for n ≥ d, then T(n) is θ($f(n)$)

The master method

- Ex: Consider the recurrence

  $T(n) = 4\,T(n/2) + n$

- $n^{log_b a} = n^{log_2 4} = n^2$
- f(n) is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$
- T(n) is $\theta(n^2)$

The master method

- Ex:

  T(n) = 2 T(n/2) + n log n

- $n^{log_b a}$ = $n^{log_2 2}$ = n
- f(n) is n log n; with k = 1, f(n) is θ(n log n)
- T(n) is θ(n log^2 n)

The master method

- Ex:

  $T(n) = T(n/3) + n$

- $n^{log_b a} = n^{log_3 1} = n^0 = 1$
- f(n) is $\Omega(n^{0+\varepsilon})$ for $\varepsilon = 1$; a f(n/b) = n/3 = (1/3)f(n)
- T(n) is $\theta(n)$

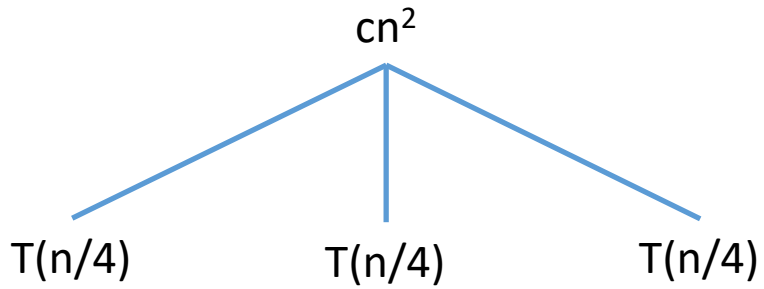
The master method

- Ex 1: $T(n) = 2^n T(n/2) + n^n$
- Ex 2: $T(n) = 2T(n/2) + n/\log n$

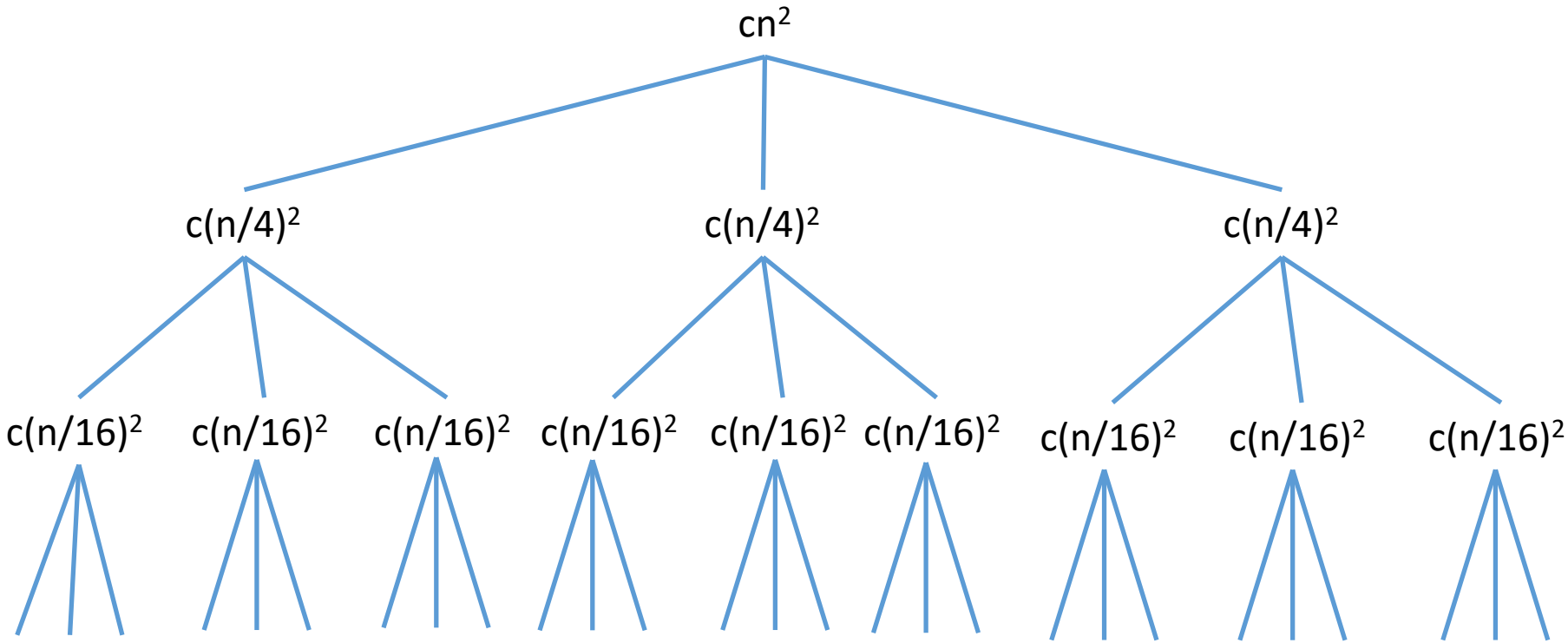
The recursive tree method

- Ex: consider the recurrence (n is an exact power of 4):

$$T(n) = \begin{cases} b & \text{if } n < 4, \\ 3T(n/4) + cn^2 & \text{otherwise} \end{cases}$$



The recursive tree method



T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) . . . T(1) T(1) T(1) T(1)

The recursive tree method

- The height of tree is: $i = \log_4 n$
- $\log_4 n + 1$ levels at depths 0, 1, 2, . . ., $\log_4 n$
- Each level has three times more nodes than previous level
- A node at depth i has cost: $c(n/4^i)^2$
- Cost per depth: $3^i\, c(n/4^i)^2 = (3/16)^i\, cn^2$
- The bottom level has $3^{\log_4 n} = n^{\log_4 3}$ nodes and each node contributes cost T(1), hence the total cost at bottom level: $\theta(n^{\log_4 3})$

The recursive tree method

$T(n) = cn^2 + (\frac{3}{16})\, cn^2 + (\frac{3}{16})^2\, cn^2 + \ldots + (\frac{3}{16})^{\log_4 n-1}\, cn^2 + \theta(n^{\log_4 3})$

$= \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i\, cn^2 + \theta(n^{\log_4 3})$

$< \sum_{i=0}^{\infty} (\frac{3}{16})^i\, cn^2 + \theta(n^{\log_4 3})$

$= \frac{1}{1-(\frac{3}{16})}\, cn^2 + \theta(n^{\log_4 3})$

$= (16/13)\, cn^2 + \theta(n^{\log_4 3})$

$= O(n^2)$

Quick sort

- Though its worst case running time is $\theta(n^2)$, the expected running time in $\theta(n \log n)$
- In place sorting algorithm

Quick sort

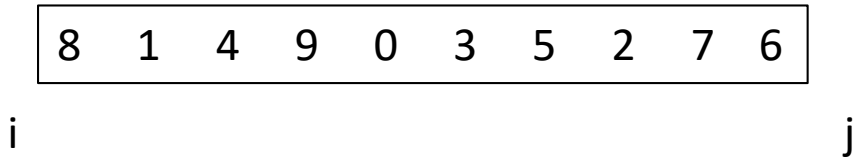
- Uses divide and conquer approach
- Divide: partition (rearrange) the given sequence A[p. . . r] into two sub-sequences A[p . . .q-1] and A[q+1 . . . r] such that all elements in A[p . . .q-1] are less than or equal to A[q] and all elements in A[q+1 . . . r] are more than or equal to A[q]; compute the index of pivot
- Conquer: Recursively sort the two sub-sequences
- Combine: Since the subarrays are already sorted no work need to be done

Partitioning Algorithm

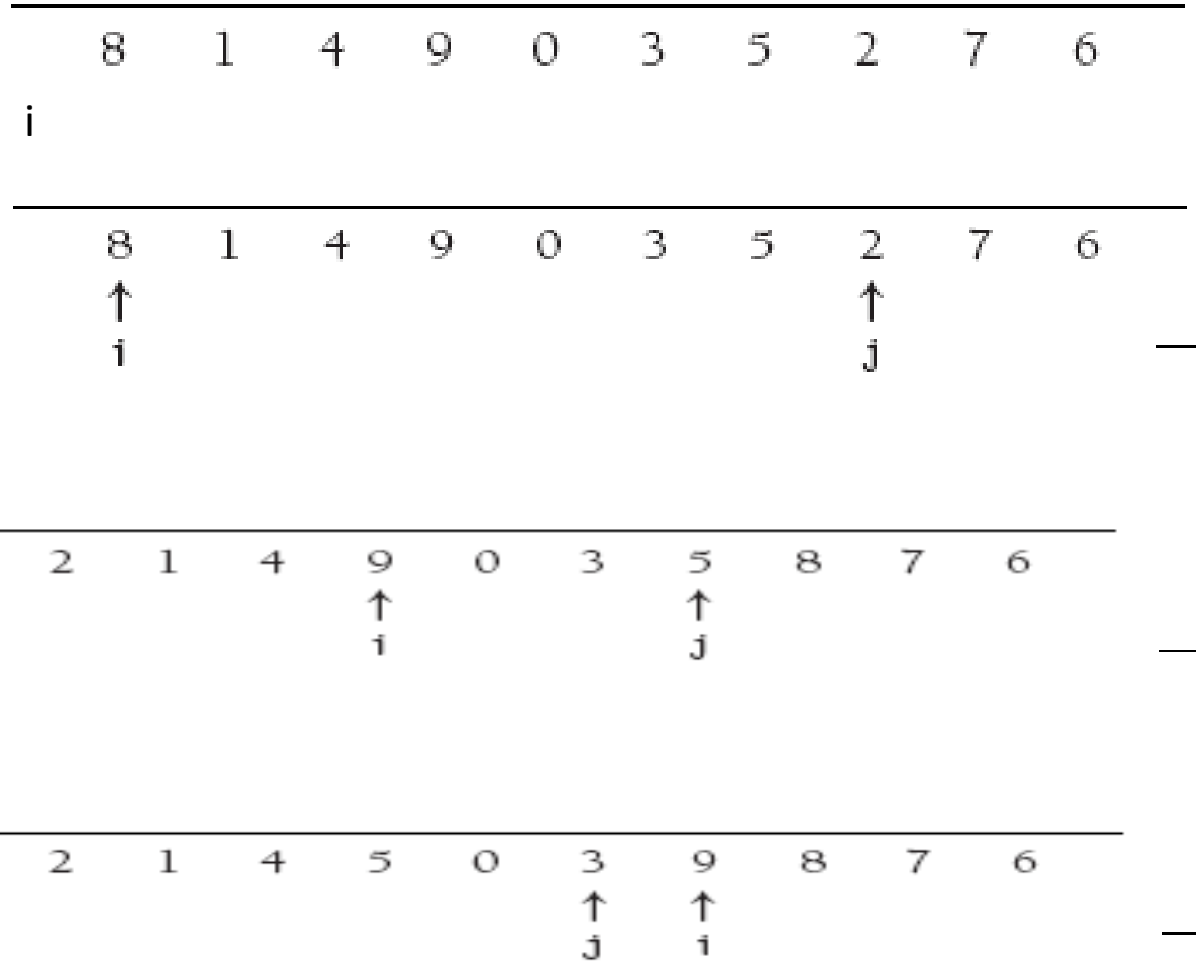
Algorithm Partition(A, p, r)

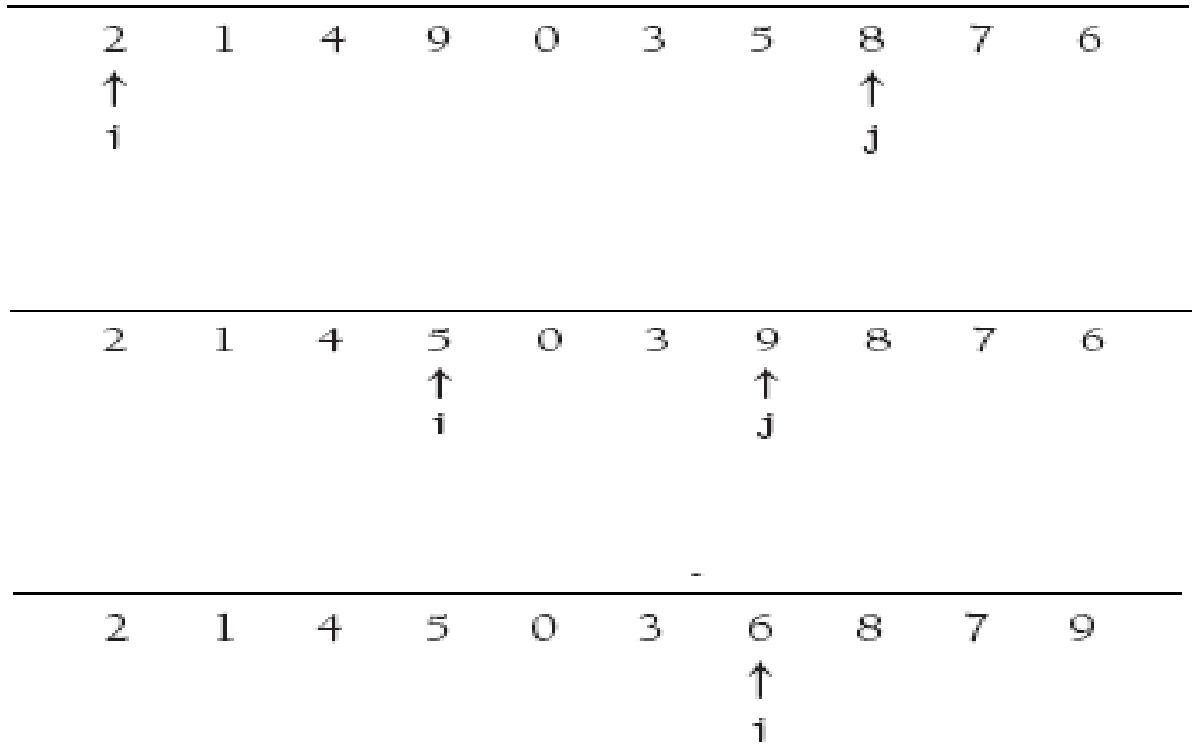
{Ensure that the algorithm works within the bounds of the input array}

```
x ← A[r]
i ← p-1
j ← r+1
for(;;)
        while(A[++i] ≤ x) { }
        while(A[--j] ≥ x) { }
        if(i < j)
                exchange(A[i], A[j])
        else
                break
exchange(A[i], A[r])
return i
```



Partitioning: Examples

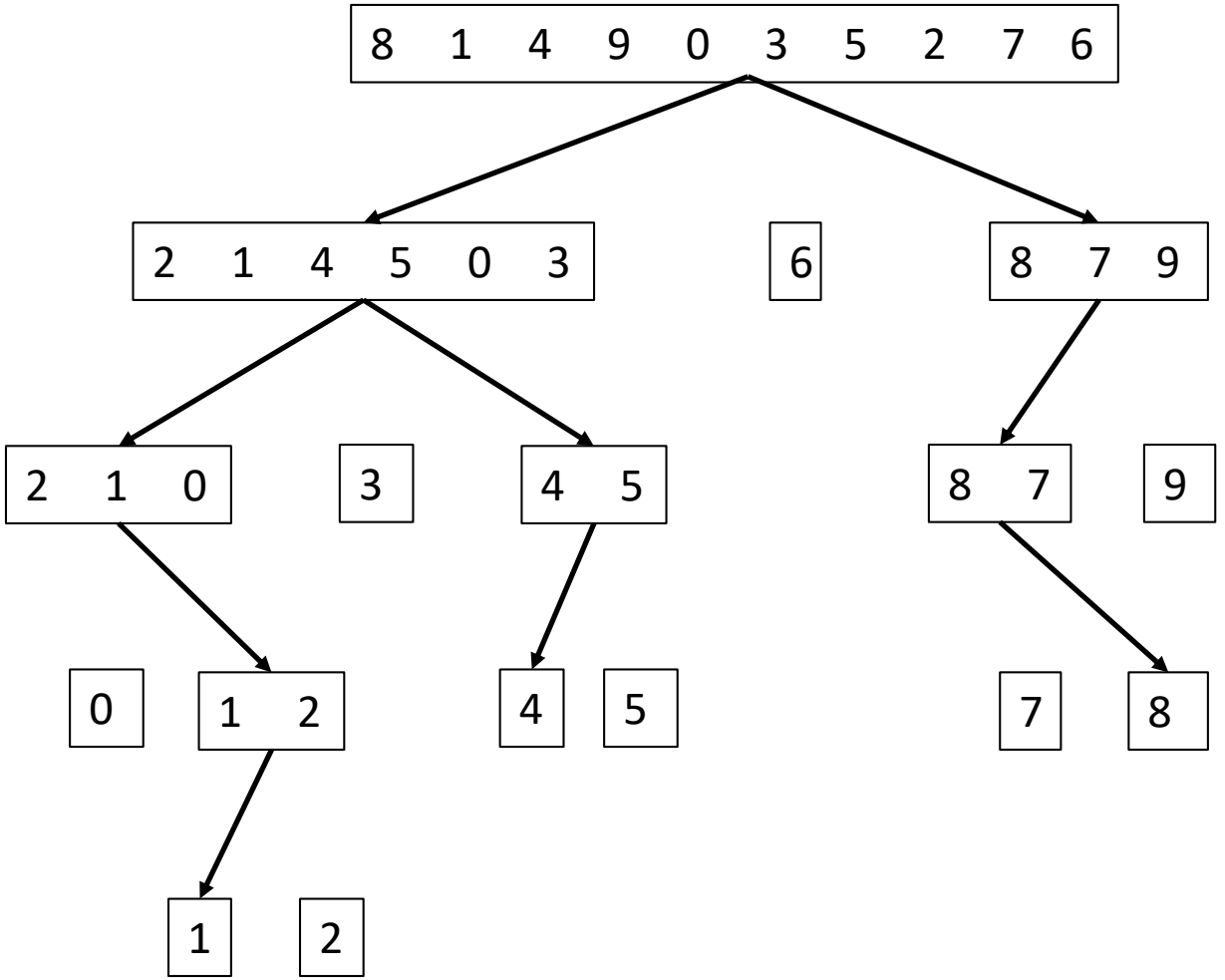




Quick sort algorithm

```
Algorithm Quick_Sort(A, p, r)
        if(p < r)
                q = Partititon(A, p, r)
                Quick_Sort(A, p, q-1)
                Quick_Sort(A, q+1, r)
```

Quick sort example



Analysis of Partitioning Algorithm

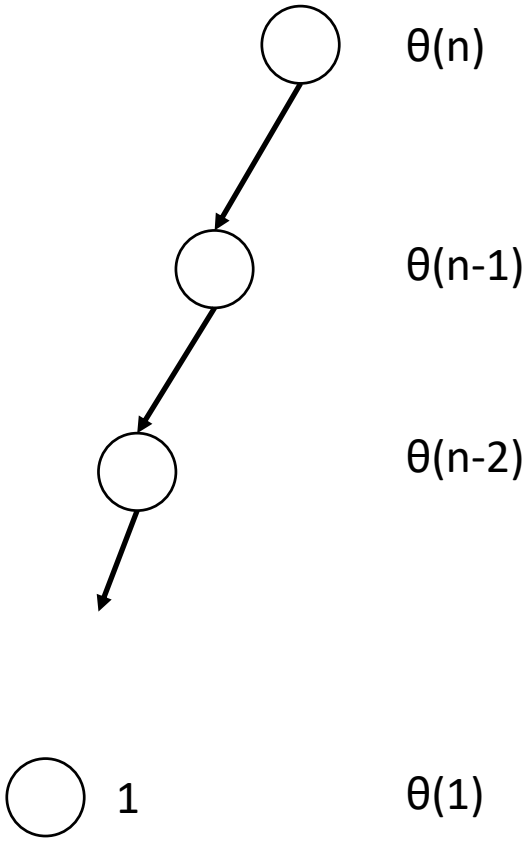
```
Algorithm Partition(A, p, r)
{Ensure that the algorithm works within the bounds of the input array}
        x ← A[r]
        i ← p-1
        j ← r+1
        for(;;)
                while(A[++i] ≤ x) { }
                while(A[--j] ≥ x) { }
                if(i < j)
                        swap(A[i], A[j])
                else
                        break
        exchange(A[i], A[r])
        return i
```

Analysis of Quick Sort: Worst case

- Assumption: All elements are distinct
- Running time depends upon how the sub-sequences are distributed
- Consider a sequence with n elements
- Partitioning produces one sub-problem with “n-1” elements and another with “0” elements
- This unbalanced partitioning arises at each recursive call

$$T(n) = T(n-1) + T(0) + \theta(n)$$

- $T(n)$ is $\theta(n^2)$



Data Structures

Analysis of Quick Sort: Worst case

- When does we encounter the worst case?
  - Input is sorted (ex: 11, 14, 25, 36, 38, 44, 48)
  - Input is reverse sorted (ex: 48, 44, 38, 36, 25, 14, 11)

- What about the performance of insertion in the above mentioned cases

Analysis of insertion sort

for j ← 1 to n-1 do	n
key ← A[j]	n-1
{insert A[j] into the sorted sequence A[0. .j-1]}	
i ← j-1	n-1
while i ≥ 0 and A[i] > key do	$\sum_{j=1}^{n-1} t_j$
A[i+1] ← A[i]	$\sum_{j=1}^{n-1} (t_j - 1)$
i - -	$\sum_{j=1}^{n-1} (tj - 1)$
A[i+1] ← key	n-1

Analysis of insertion sort

$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} t_j + \sum_{j=1}^{n-1}(t_j - 1) + \sum_{j=1}^{n-1}(t_j - 1) + (n-1)$

$= n + 3(n-1) + (n-1)n/2 + (n-1)(n-2)$

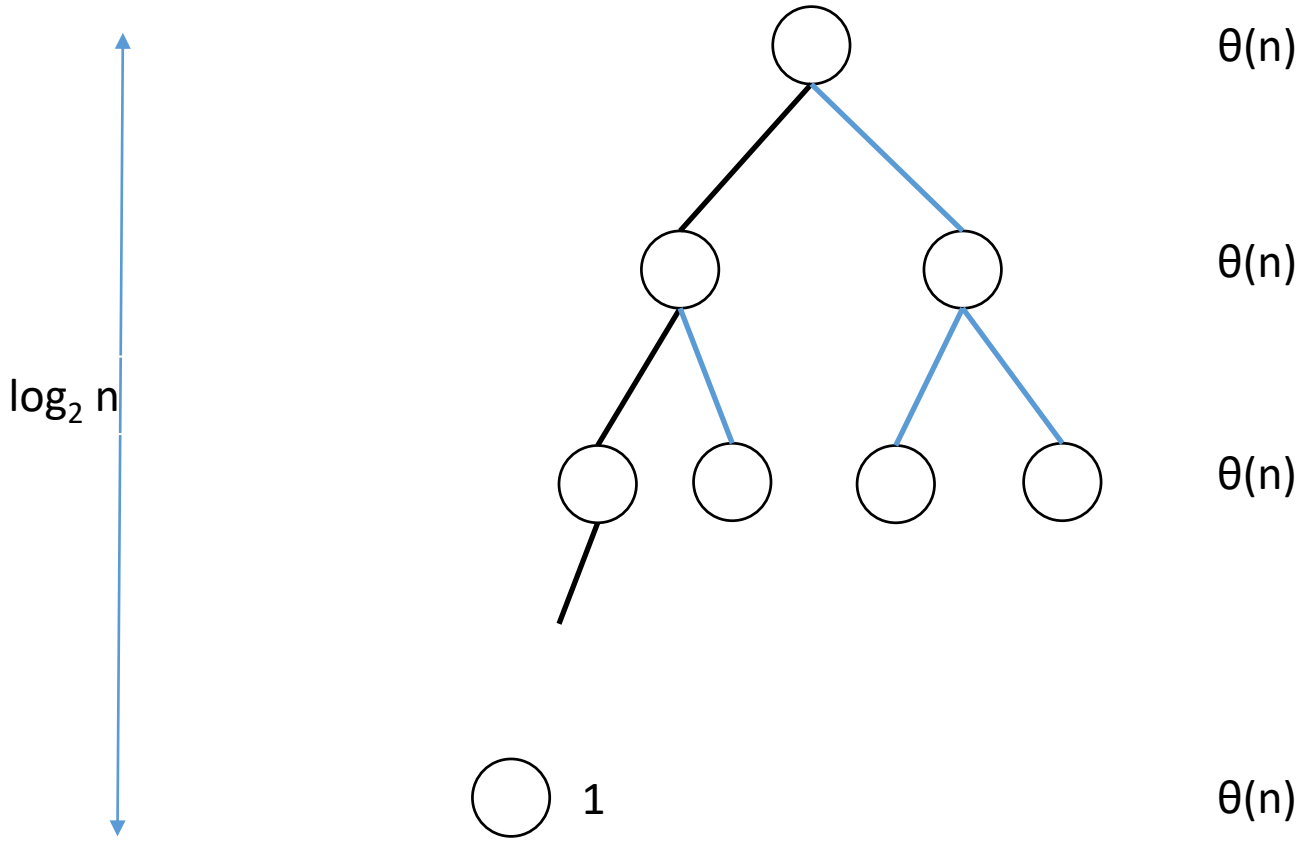
$= an^2 + bn + c$

Analysis of Quick Sort: Best case

- Consider a sequence with n elements
- The partition step splits the given sequence evenly
- $\lfloor n/2 \rfloor, \lceil n/2 \rceil - 1$

  $T(n) = 2T(n/2) + \theta(n)$
- The height of the recursive tree of quick sort is: $\theta(\log n)$

Analysis of Quick Sort: Best case



Analysis of Quick Sort: Best case

- Guess is: T(n) is θ(n log n), that is, $T(n) \leq c_1 n \log n$ and $T(n) \geq c_2 n \log n$
- Consider the case $T(n) \leq c_1 n \log n$

$T(n) \leq 2c_1 (n/2) \log (n/2) + \theta(n)$

$\leq c_1 n \log (n/2) + \theta(n)$

$\leq c_1 n \log n - c_1 n + \theta(n)$

$\leq c_1 n \log n$

Analysis of Quick Sort: Best case

- Consider the case $T(n) \geq c_2 n \log n$

$T(n) \geq 2c_2 (n/2) \log (n/2) + \theta(n)$

$\geq c_2 n \log (n/2) + \theta(n)$

$\geq c_2 n \log n - c_2 n + \theta(n)$

$\geq c_2 n \log n$

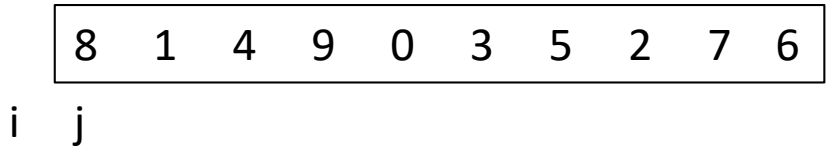
Pivot selection

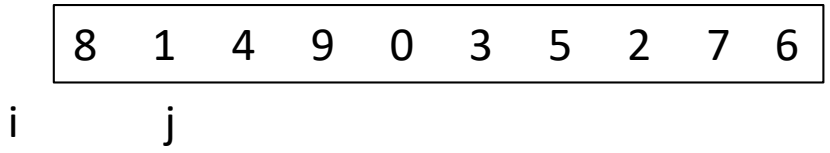
- Popular choice: either the last element or the first element
  - Creates problem when input is in sorted or reverse sorted order
- Safe option: choose the pivot randomly (expensive)
- Median: Median of the array (and median of the first, middle and the last element in the given sequence)

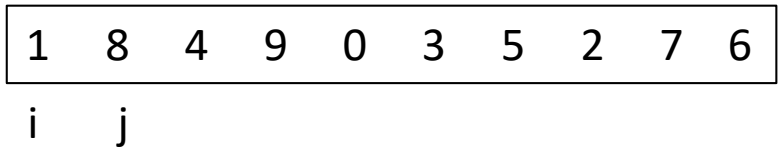
Quick sort: Analysis

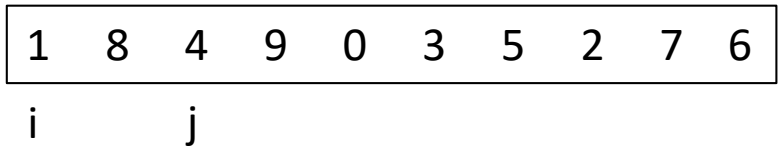
```
Algorithm Alt(A, p, r)
        x ← A[r]
        i ← p-1
        for j ← p to r-1
                if(A[j] ≤ x)
                        i ← i +1
                        exchange A[i] and A[j]
        exchange A[i+1] and A[r]
        return i+1
```

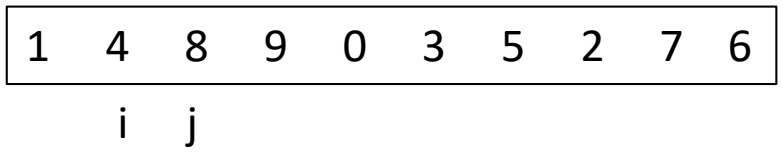
Partitioning

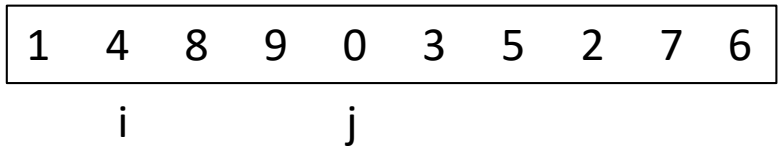


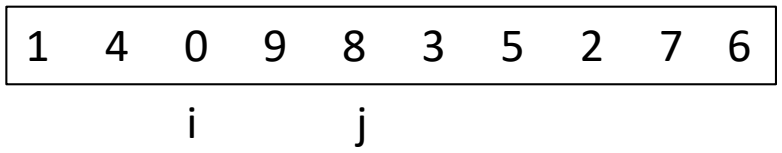


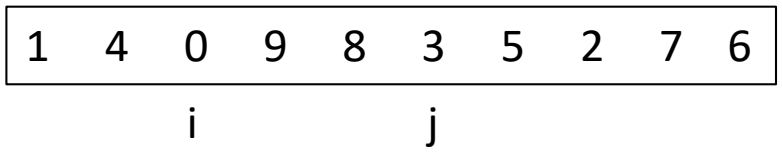


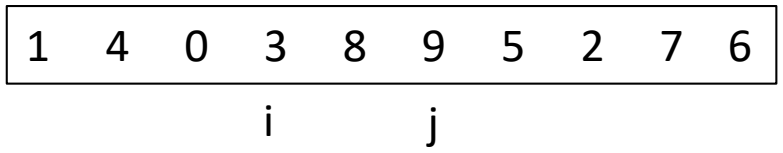


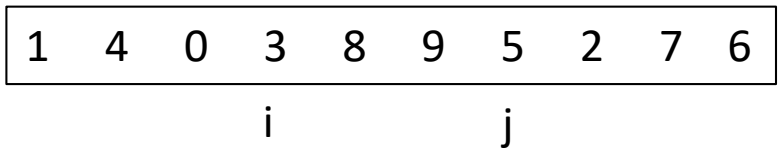




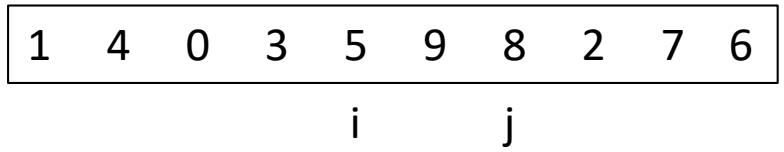


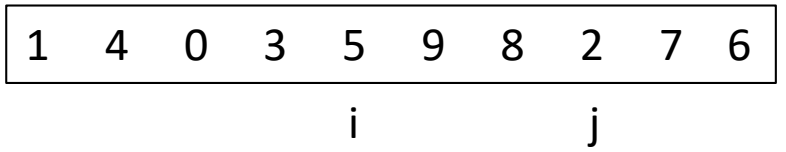


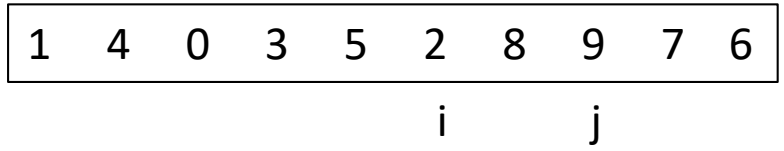


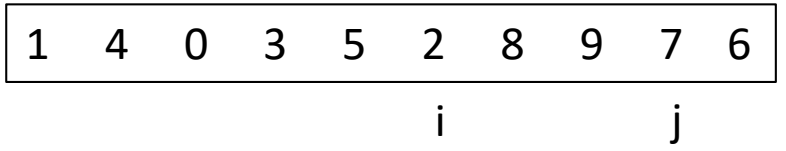


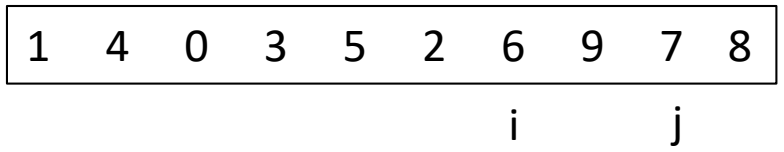
Partitioning









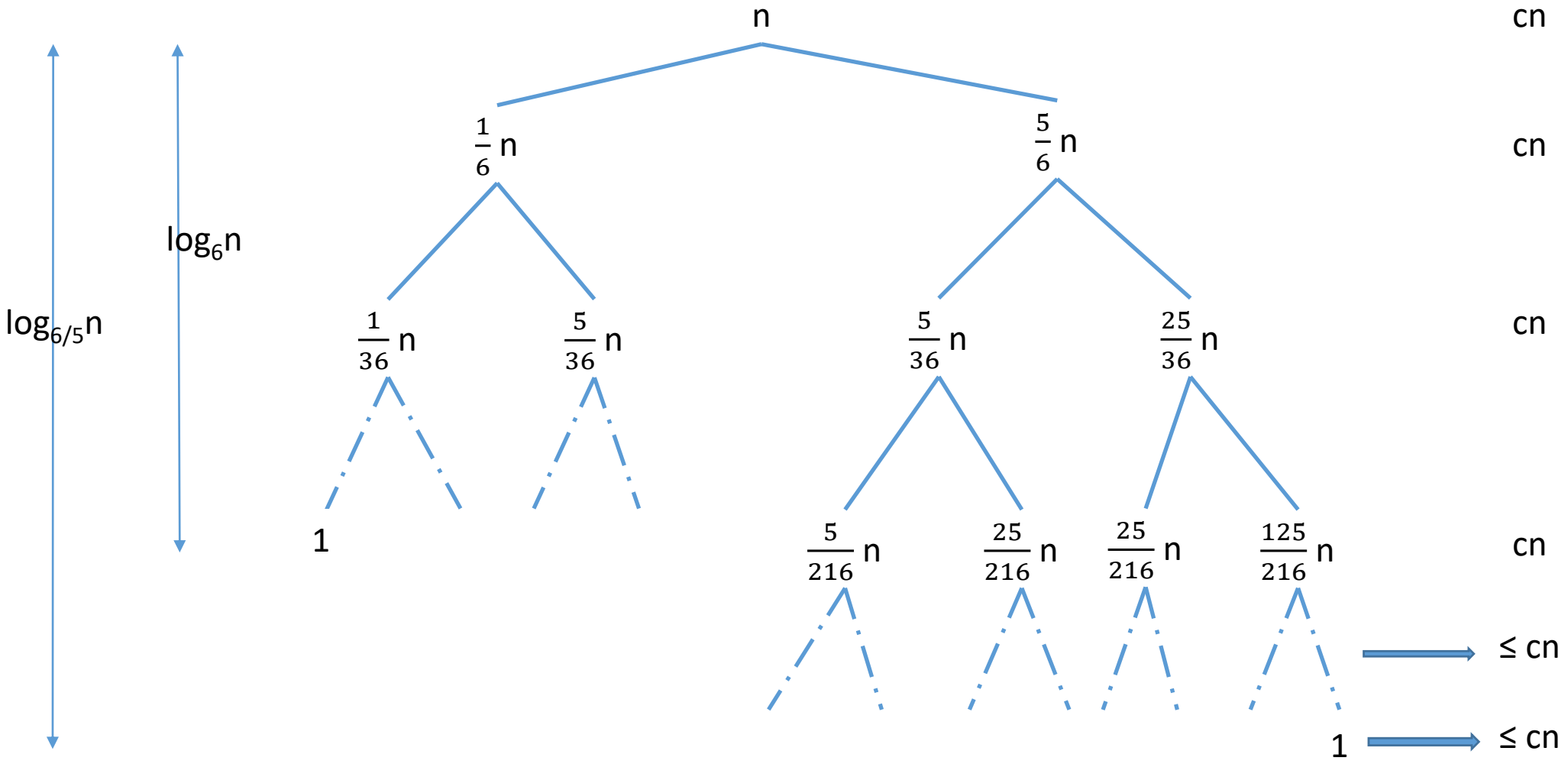


Partitioning Algorithm

```
Partition(A, p, r)
        x ← A[r]
        i ← p-1
        j ← r+1
        for(;;)
                while(A[++i] ≤ x) { }
                while(A[--j] ≥ x) { }
                if(i < j)
                        swap(A[i], A[j])
                else
                        break
        exchange(A[i], A[r])
        return i
```

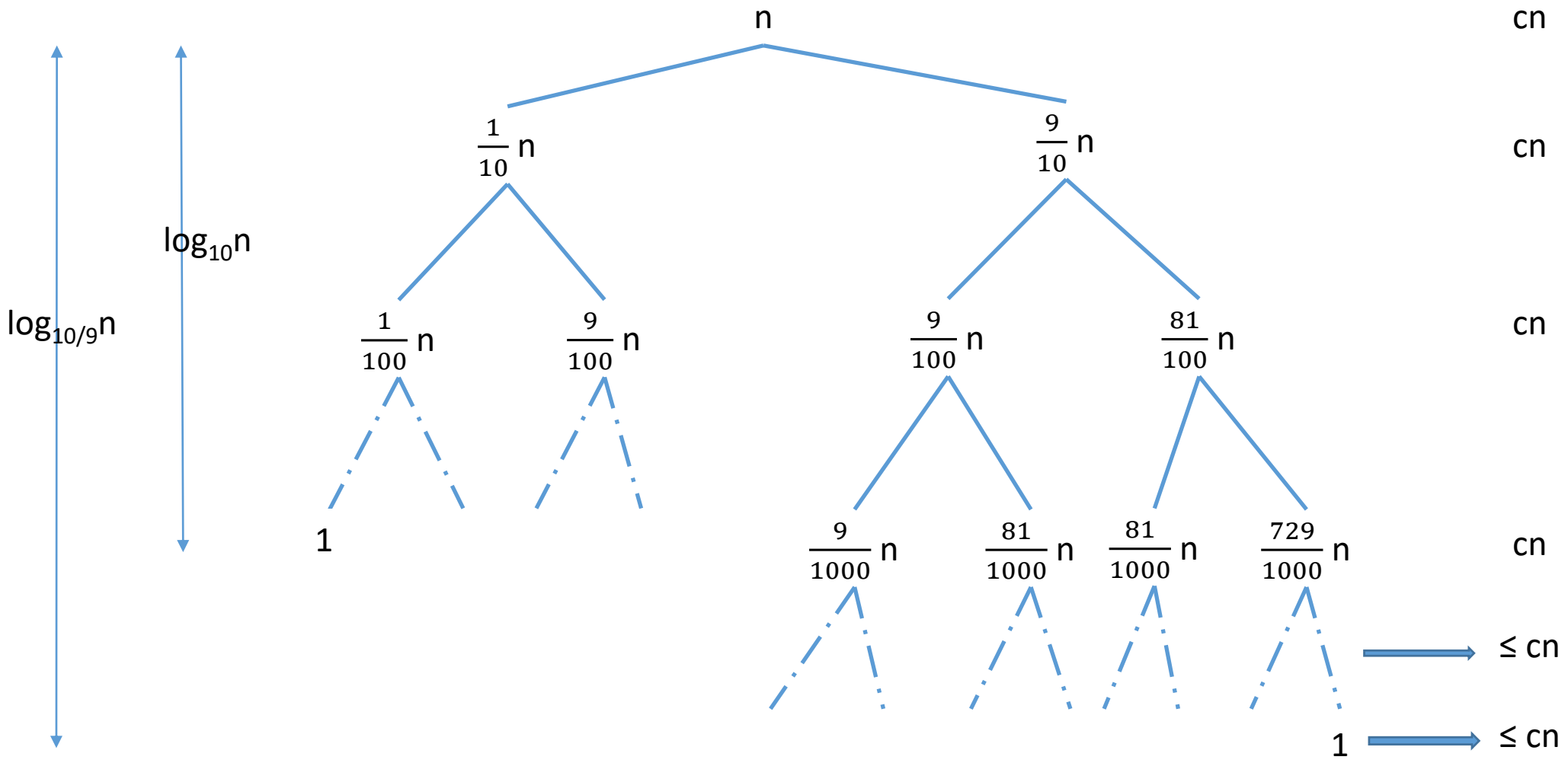
Analysis of Quick Sort

- Assume that the partitioning algorithm produces 5-to-1 proportional split: T(n) = T(n/6) + T(5n/6) + cn



Analysis of Quick Sort

- Assume that the partitioning algorithm produces 9-to-1 proportional split: T(n) = T(n/10) + T(9n/10) + cn



Data Structures

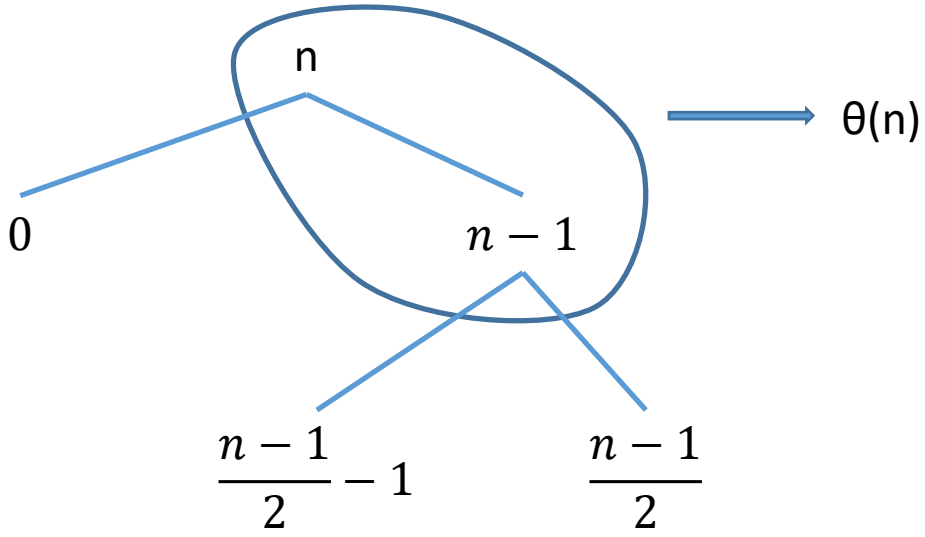
Analysis of Quick Sort: average case

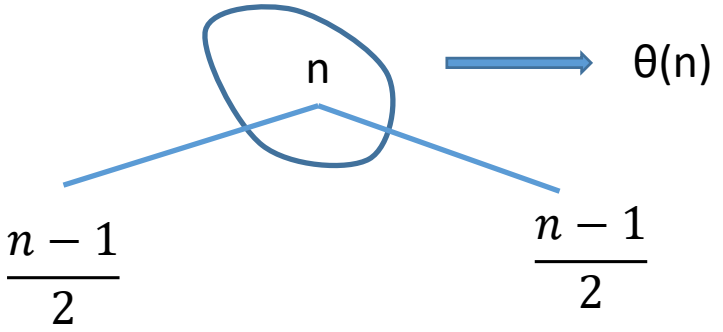
- The behaviour of quick sort depends upon the relative ordering of the elements in the input array
- All permutations of input numbers are equally likely (assumption)
- On a random input array, some of the splits will be reasonably balanced and some are fairly unbalanced
- In the average case, Partition procedure produces a mix of good (best-case splits) and bad splits (worst-case splits)
- In a recursion tree for an average-case execution of Partition, the good and bad splits are distributed randomly throughout the tree

Analysis of Quick Sort: average case

- Suppose that the good and bad splits alternate levels in the tree
- A bad split happens at the root which produces two sub arrays of sizes “0” and “n-1”
- At the next level a good split happens which produces two sub arrays of sizes “(n-1)/2-1” and “(n-1)/2”
- The combination of a bad split followed by a good split produces three sub arrays of sizes 0, (n-1)/2-1 and (n-1)/2
- The partitioning cost of these splits is: $\theta(n) + \theta(n-1) = \theta(n)$

Analysis of Quick Sort





Randomized quick sort

- In a practical situation all permutations are not equally likely
- Can add randomization to an algorithm to obtain a good expected performance over all inputs
- The resulting algorithm is called randomized quick sort
- A randomization technique called “random sampling” is used
- Use a randomly chosen element from given subarray as the pivot instead of the rightmost element
- The input array is expected to get split into reasonably balanced sets on average

Randomized quick sort

```
Randomized_Partition(A, p, r)
        i ← RANDOM(p, r)
        exchange A[r] with A[i]
        return Partition(A, p, r)
Randomized_QuickSort(A, p, r)
        if(p < r)
                q ← Randomized_Partitiiton(A, p, r)
                Randomized_QuickSort(A, p, q-1)
                Randomized_QuickSort(A, q+1, r)
```

Randomized quick sort: Worst-case

$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \theta(n)$

Use guess-and-test method: $T(n) \le cn^2$ for some constant c (hypothesis)

$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \theta(n)$

$= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \theta(n)$

$(q^2 + (n-q-1)^2)$ achieves the maximum at both the end points of the range

$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$

$T(n) \le cn^2 - c(2n - 1) + \theta(n)$

$\le cn^2$

Randomized quick sort: Worst-case

$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \theta(n)$

Use guess-and-test method: $T(n) \ge cn^2$ for some constant c (hypothesis)

$T(n) \ge \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \theta(n)$

$= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \theta(n)$

$(q^2 + (n-q-1)^2)$ achieves the maximum at both the end points of the range

$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) = (n-1)^2 = n^2 - 2n + 1$

$T(n) \ge cn^2 - c(2n-1) + \theta(n)$

$\ge cn^2$, where c is chosen so that $\theta(n)$ dominates $c(2n-1)$

A few basics

- Consider a sample space S and an event A
- Indicator random variable associated with event A is denoted as I{A} and defined as:
  - $I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$

A few basics

- The expected value or the expectation of a discrete random variable is:

$$E[X] = \sum_x x.\Pr\{X = x\}$$

- $I\{A\} = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$

- Linearity of expectation: The expectation of sum of two random variables is the sum of their expectations

$$E[X + Y] = E[X] + E[Y]$$

A few basics

Experiment: flipping a fair coin

$S = \{H, T\}$; $Pr\{H\} = ½$ and $Pr\{T\} = ½$

H: the coin coming up with heads

X_H : indicator random variable associated with H

$X_H = I\{H\}$

$$I\{H\} = \begin{cases} 1 & \text{if H occurs} \\ 0 & \text{if T occurs} \end{cases}$$

A few basics

The expected number of heads in one flip of coin is the expected value of our indicator X_H:

$E[X_H] = E[I\{H\}]$

$= 1.Pr\{H\} + 0.\ Pr\{T\}$

$= 1.1/2 + 0.1/2$

$=1/2$

Lemma: Given a sample space S and an event A in S, let $X_A = I\{A\}$. Then

$E[X_A] = Pr\{A\}$

Quick sort: Analysis

```
Algorithm Partition(A, p, r)
        x ← A[r]
        i ← p-1
        for j ← p to r-1
                if(A[j] ≤ x)
                        i ← i +1
                        exchange A[i] and A[j]
        exchange A[i+1] and A[r]
        return i+1
```

Randomized quick sort: Expected running time

- Randomized quick sort works similar to quick sort except for pivot selection
- Analyse Randomized Quick_Sort by discussing Quick_Sort and Partition algorithms (randomly selected pivot)
- Consider an array of n distinct elements
- The running time of Quick_Sort is dominated by the time spent in Partition procedure
- How many times an element is selected as pivot?
- Observation 1: An element selected as pivot never included in the future recursive calls to Quick_Sort and Partition
- There can be at most n calls to Partition

Randomized quick sort: Expected running time

- One call to Partition:
  - constant amount of time and
  - Amount of time proportional to number of iterations of the **for** loop
- In each iteration of **for** loop, the pivot is compared with an element in the array (pivot-array element comparison)
- What is the total time spent in the **for** loop over all calls to Partition procedure?
- By counting the number of times the pivot is compared to an array element, we can bound the total time spent in the **for** loop

Randomized quick sort: Expected running time

Lemma: Let X be the number of pivot-array element comparisons performed over the entire execution of Quick_Sort on an n-element array. Then the running time of Quick_Sort is O(n + X)

Proof: The algorithm makes at most n calls to Partition procedure

Each call does a constant amount of work and executes the **for** loop some number of times

Each iteration of **for** loop performs one pivot-array element comparison

We have to compute “X”, the total number of comparisons performed over all calls to Partition

Randomized quick sort: Expected running time

- Rename the elements in array A as $z_1, z_2, \ldots, z_n$, where z_i is the i^{th} smallest element in A
- $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between z_i and z_j inclusive
- Observation 2: each pair of elements are compared at most once, why?
- Define indicator random variable as:

  $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

  (during entire execution of the algorithm)
- Since each pair of elements is compared at most once,

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Randomized quick sort: Expected running time

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \, is \; compared \; to \; z_j\}$$

- We have to compute the quantity, $\Pr\{z_i \, is \; compared \; to \; z_j\}$
- Consider an input array A = {41, 11, 21, 51, 81, 61, 31, 71, 101, 91}
- Assume that the first call to Partition separates this array into two sets: {41, 11, 21, 51, 31} and {81, 71, 101, 91}
- An element from either of these sets will ever be compared with the elements in the other set?

Data Structures

Randomized quick sort: Expected running time

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \, is \; compared \; to \; z_j\}$$

- We have to compute the quantity, $\Pr\{z_i \, is \; compared \; to \; z_j\}$
- Consider an input array A = {41, 11, 21, 51, 81, 61, 31, 71, 101, 91}
- Assume that the first call to Partition separates this array into two sets: {41, 11, 21, 51, 31} and {81, 71, 101, 91}
- An element from either of these sets will ever be compared with the elements in the other set?

Randomized quick sort: Expected running time

- Consider a set Z_{ij}
- If an element "x" s.t. $z_i < x < z_j$ is selected as the pivot, then can we compare z_i and z_j at any subsequent time?
- If z_i is chosen as the pivot, then z_i will be compared to all elements in Z_{ij} except for itself
- If z_j is chosen as the pivot, then z_j will be compared to all elements in Z_{ij} except for itself
- Observation 3:

Elements z_i and z_j are compared if the first element to be chosen as pivot from Z_{ij} is either z_i or z_j

Randomized quick sort: Expected running time

- Till the point we select an element from Z_{ij} as the pivot, all elements in Z_{ij} are in the same partition
- A = {41, 11, 21, 51, 81, 61, 31, 71, 101, 91}
- First two smallest elements: Z_{12} = {11, 21}
- A_1 = {41, 11, 21, 51, 31}; A_2 = {81, 71, 101, 91}
- A_{11} = {11, 21}; A_{12} = {41, 51}
- A_{111} = {11}
- Any elements from Z_{ij} is equally likely to be selected as the pivot
- There are j-i+1 elements in Z_{ij}
- The probability that any given element is the first one chosen as pivot is 1/(j-i+1)

Randomized quick sort: Expected running time

- $\Pr\{z_i \text{ is compared to } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$

  $= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} +$

  $\Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$

  $= 1/(j-i+1) + 1/(j-i+1)$

  $= 2/(j-i+1)$

Randomized quick sort: Expected running time

- Using Pr{z_i is compared to z_j},

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(j-i+1)$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2/(k+1)$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} 2/k$$

$$= \sum_{i=1}^{n-1} \mathrm{O}(\log n)$$

E[X] is O(n log n)

- Using Randomized_Partition, the expected running time of quick sort algorithm is O(n log n) when the elements are distinct

Alternate analysis of expected running time

- Analyse the running time in terms of the expected running time of each individual recursive calls to randomised quicksort
- An array of n distinct elements
- The probability of selecting an element as pivot is: 1/n
- X_i = I{ith smallest element is selected as pivot}

$$= \begin{cases} 1 & \text{if ith smallest element is selected as pivot} \\ 0 & \text{otherwise} \end{cases}$$

- $E[X_i] = 1/n$

Alternate analysis of expected running time

- Suppose that ith smallest element is selected as the pivot
- The input array gets divided into two subarrays of size (i-1) and (n-i)
- The running time is:

$$T(n) = X_i(T(i-1) + T(n-i) + \theta(n)$$

- Expectation over all events and linearity of expectation

$$E[T(n)] = E[\sum_{i=1}^{n} X_i(T(i-1) + T(n-i) + \theta(n))]$$
$$= \sum_{i=1}^{n} E[X_i(T(i-1) + T(n-i) + \theta(n))]$$
$$= \sum_{i=1}^{n} E[X_i](E[T(i-1)] + E[T(n-i)] + \theta(n))$$
$$= \sum_{i=1}^{n} (E[T(i-1)] + E[T(n-i)] + \theta(n))/n$$

Alternate analysis of expected running time

$$E[T(n)] = \theta(n) + 1/n \sum_{i=1}^{n} (E[T(i-1)] + E[T(n-i)])$$

$$= \theta(n) + \frac{1}{n} \left(\sum_{i=1}^{n} E[T(i-1)] + \sum_{i=1}^{n} E[T(n-i)]\right)$$

$$= \theta(n) + \frac{1}{n} \left(\sum_{i=1}^{n} E[T(i-1)] + \sum_{i=1}^{n} E[T(i-1)]\right)$$

$$= \theta(n) + \frac{2}{n} \sum_{i=1}^{n} E[T(i-1)]$$

$$= \theta(n) + \frac{2}{n} \sum_{i=0}^{n-1} E[T(i)]$$

$$= \theta(n) + \frac{2}{n} \sum_{i=2}^{n-1} E[T(i)]$$

Alternate analysis of expected running time

$$E[T(n)] = \theta(n) + \frac{2}{n}\sum_{i=2}^{n-1} E[T(i)]$$

$$E[T(n)] = \frac{2}{n}\sum_{i=2}^{n-1} E[T(i)] + cn$$

$$nE[T(n)] = 2\sum_{i=2}^{n-1} E[T(i)] + cn^2$$

$$(n-1)\,E[T(n-1)] = 2\sum_{i=2}^{n-2} E[T(i)] + c(n-1)^2$$

Take difference:

$$nE[T(n)] - (n-1)\,E[T(n-1)] = 2E[T(n-1)] + 2cn - c$$

$$nE[T(n)] = (n+1)E[T(n-1)] + 2cn$$

Divide by n(n+1):

$$\frac{E[T(n)]}{n+1} = \frac{E[T(n-1)]}{n} + \frac{2c}{n+1}$$

Alternate analysis of expected running time

$$\frac{E[T(n)]}{n+1} = \frac{E[T(n-1)]}{n} + \frac{2c}{n+1}$$

$$\frac{E[T(n-1)]}{n} = \frac{E[T(n-2)]}{n-1} + \frac{2c}{n}$$

$$\frac{E[T(n-2)]}{n-1} = \frac{E[T(n-3)]}{n-2} + \frac{2c}{n-1}$$

.

.

.

$$\frac{E[T(2)]}{3} = \frac{E[T(1)]}{2} + \frac{2c}{3}$$

Adding these equations:

$$\frac{E[T(n)]}{n+1} = \frac{E[T(1)]}{2} + 2c\sum_{i=3}^{n+1} \frac{1}{i}$$

Alternate analysis of expected running time

$$\frac{E[T(n)]}{n+1} = \frac{E[T(1)]}{2} + 2c\sum_{i=1}^{n+1} \frac{1}{i} - \frac{3}{2}$$

E[T(n)] is O(n log n)

Space complexity

- It is a function describing the amount of memory space an algorithms takes in terms of input size
- Can use natural units to measure the space requirement
- Number of integers used, number of fixed size structures
- The function is independent of bytes needed to represent a unit

Algorithm sum(x, y, z)

    r ← x + y + z

    return r

Space complexity

```
Algorithm sum1(A, n)
        r ← 0
        for i ← 0 to n-1 do
                r ← (r + A[i])
        return r
```

Space complexity

```
Algorithm matrixAdd(A, B, n)
        for i ← 0 to n-1 do
                for j ← 0 to n-1 do
                        A[i,j] ← (A[i,j] + B[i,j])
```

Quick sort algorithm

Initial call: Quick_Sort(A, 0, n-1)

```
Quick_Sort(A, p, r)
        if(p < r)
                q = Partititon(A, p, r)
                Quick_Sort(A, p, q-1)
                Quick_Sort(A, q+1, r)
```

Another version of quicksort

- The second recursive call is not really necessary
- Can be avoided using an iterative control structure
- This technique is called tail recursion

```
Tail_Recursive_Quicksort(A, p, r)
        while p < r
                q ← Partition(A, p, r)
                Tail_Recursive_Quicksort(A, p, q-1)
                p ← q+1
```

Another version of quicksort

- Compilers use a stack to execute recursive calls
- This stack contains pertinent information including the parameter values for each recursive call
- Recent information is at the top
- When a procedure is called, its information is pushed onto stack
- When a procedure is terminated, its information is popped from stack
- Assume that the array parameters are represented by pointers
- The information for each procedure call: O(1)
- Stack depth: the maximum amount of stack space used at any time during a computation

Another version of quicksort

Initial call: Tail_Recursive_Quicksort(A, 0, n-1)

```
Tail_Recursive_Quicksort(A, p, r)
        while p < r
                q ← Partition(A, p, r)
                Tail_Recursive_Quicksort(A, p, q-1)
                p ← q+1
```

When does the stack depth is θ(n) on an n-element input array?

Another version of quicksort

- How can we modify the code so that worst-case stack depth is θ(log n)

```
Algorithm Tail_Recursive_Quicksort1(A, p, r)
        while p < r
                q ← Partition(A, p, r)
                if q < ⌊(p + r)/2⌋
                        Tail_Recursive_Quicksort1(A, p, q-1)
                        p ← q+1
                else
                        Tail_Recursive_Quicksort1(A, q+1, r)
                        r ← q-1
```

Data Structures

Comparison based sorting: Lower bound

- Insertion sort, merge sort, and quick sort
- Primitive operation used is: Comparison
- Consider a sequence S = (a_1, a_1, . . ., a_n) of distinct elements
- Compare two elements, say a_i and a_j (outcome: “yes” or “no”)
  - Perform few steps
  - Compare another pair of elements

Comparison based sorting: Lower bound

- Insertion sort:
  - Compares the first element (say y) from the unsorted part to the last element (say x) in the sorted part
  - Change the position of the last element in the sorted if y < x
  - Compares y to the previous element of x
- Merge sort:
  - Compares the first two elements in the sorted sub-sequences
  - Appends the smallest to the end of the result sequence
- Quick sort:
  - Picks the pivot and place “i“ and “j” at appropriate positions
  - Increments i, and compares the element at ith position to the pivot
  - Decrements j, and compares the element at jth position to the pivot
  - Performs required actions

Comparison based sorting: Lower bound

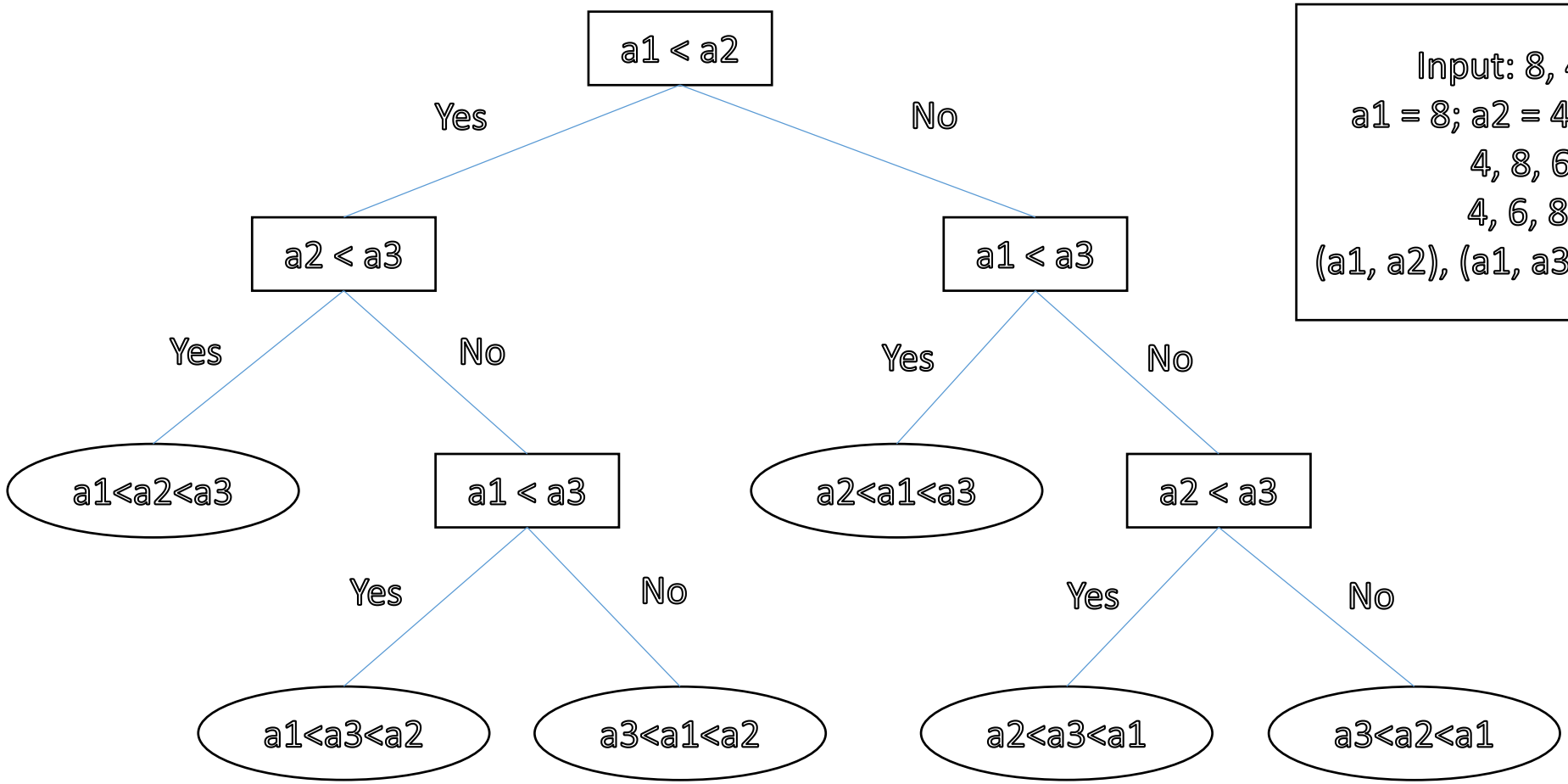
- Consider an input array (8, 4, 6)
- (4, 8, 6)
- (4, 6, 8)
- The comparisons performed are: (4, 8), (6, 8), (4, 6)
- Consider another permutation of input array, say (8, 6, 4)
- (6, 8, 4)
- (4, 6, 8)
- The comparisons performed are: (6, 8), (4, 8), (4, 6)

Comparison based sorting: Lower bound

- A comparison based sorting algorithm can be represented in the form of a decision tree
- All possible sequences of comparisons performed on an n-element sequence (a_1, a_2, . . ., a_n)
- Each internal node represents a comparison
- Associate with each external node v the permutation that causes the sorting algorithm to end up in v
- Each possible input permutation causes the algorithm to execute a series of comparisons (traverse a path) and end up at some external node

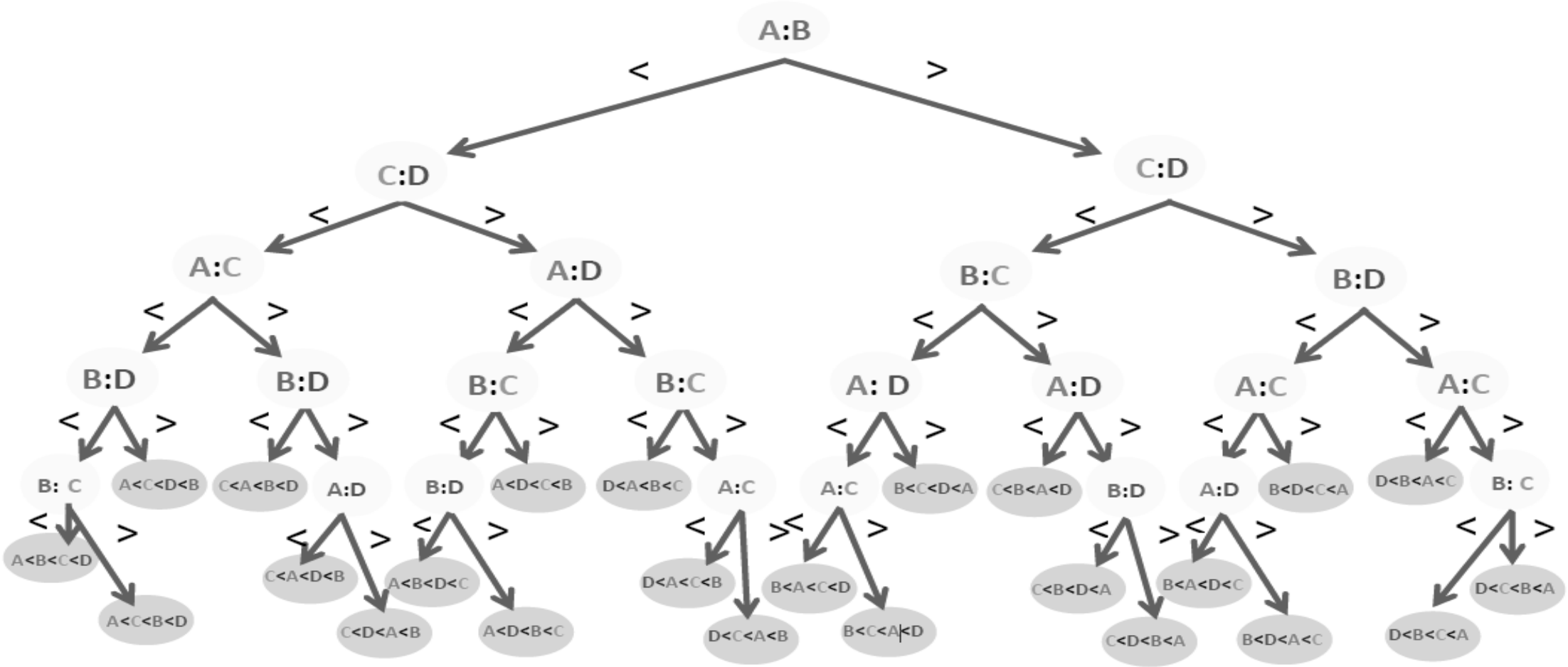
Comparison based sorting: Lower bound

- Consider a general sequence of three elements (a1, a2, a3)



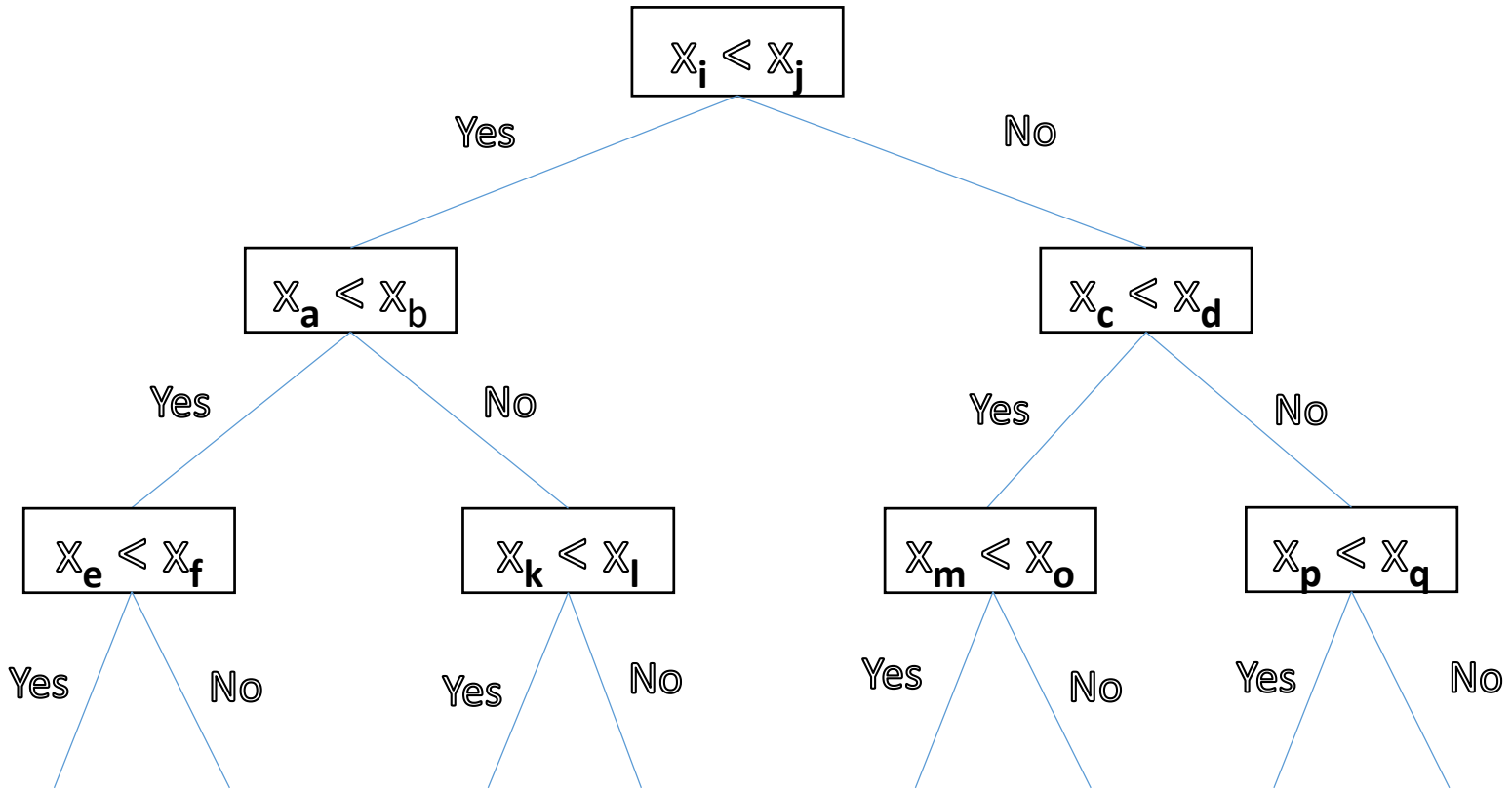
Comparison based sorting: Lower bound

- Consider a general sequence of four elements (A, B, C, D):



Comparison based sorting: Lower bound

- Consider a general sequence (S) of n elements ($x_1, x_2, \ldots, x_n$)



Comparison based sorting: Lower bound

- There exists a 1-1 correspondence between input permutations and external nodes
- Consider two permutations, P1 and P2, of $S = (x_1, x_2, \ldots, x_n)$
- Assume that P1 and P2 end up at the eternal node v
- x_i appears before x_j in P1 and x_i appears after x_j in P2
- The permutation associated with v is P3 of S where x_i appears either before or after x_j

Comparison based sorting: Lower bound

- **Theorem:**

  The running time of any comparison-based algorithm for sorting some n-element input sequence ($S = (x_1, x_2, \ldots, x_n)$) is $\Omega(n \log n)$ in the worst-case

- **Proof:**

  The running time must be greater than or equal to the height of its decision tree

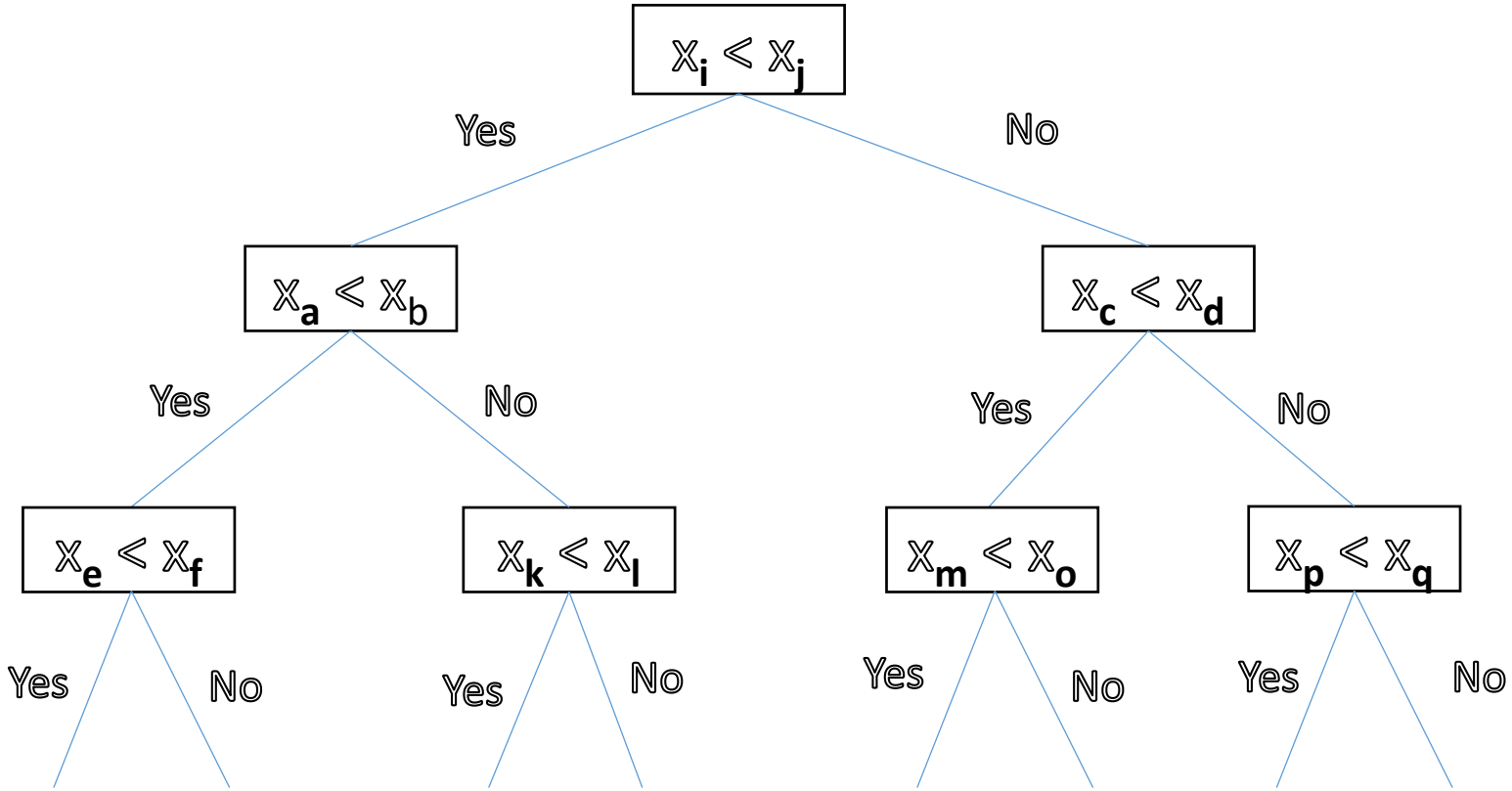
  Each external node must be associated with one permutation of the input

  Each permutation must result in a different external node

  The number of permutations of S are: n!

  There exists “n!” external nodes in the decision tree

Comparison based sorting: Lower bound



n!

Comparison based sorting: Lower bound

- **Proof (Contd):**

Can we say any thing about the height of a binary tree in term of number of leaves?

$h \geq \lceil \log(m) \rceil$ (m is the number of leaves)

Height of decision tree is: log(n!)

$\log(n!) \geq \log(n/2)^{n/2}$

$= n/2 \log(n/2)$

is $\Omega(n \log n)$

Bucket Sort

- Can run asymptotically faster than O(n log n), but with special assumptions
- Consider a sequence S of n items
- Item: (key, element)
- The keys are in the range [0, . . ., N-1] for some $N \geq 2$
- S should be sorted according to the keys
- If N is O(n), then we can sort S in O(n) time
- Restrictive assumption about the format of elements

Bucket sort

- Not based on comparisons
- Uses an array (of size N) of buckets to categorize the items based on keys
- Keys are used as indices (0, . . ., N-1) into the bucket array B
- An item with key “k” is placed in bucket B[k] (sequence of items with key k)
- Place the items back into S in sorted order, how?
- Enumerate the items in the buckets B[0], B[1], . . ., B[N-1] in order

Bucket sort

Algorithm Bucket_Sort(S)

Input: Sequence S of items with integer keys in the range 0, . . ., N-1

Output: Sequence S sorted in nondecreasing order of the keys

```
let B be an array of N sequences, each of which is initially empty
for each item x in S do
        let k be the key of x
        remove x from S and insert x at the end of B[k]
for i ← 0 to N-1 do
        for each item x in bucket B[i] do
                remove x from B[i] and insert it at the end of S
```

Bucket sort

- Time complexity?
- Space complexity?
- What happens as N increases compared to n?

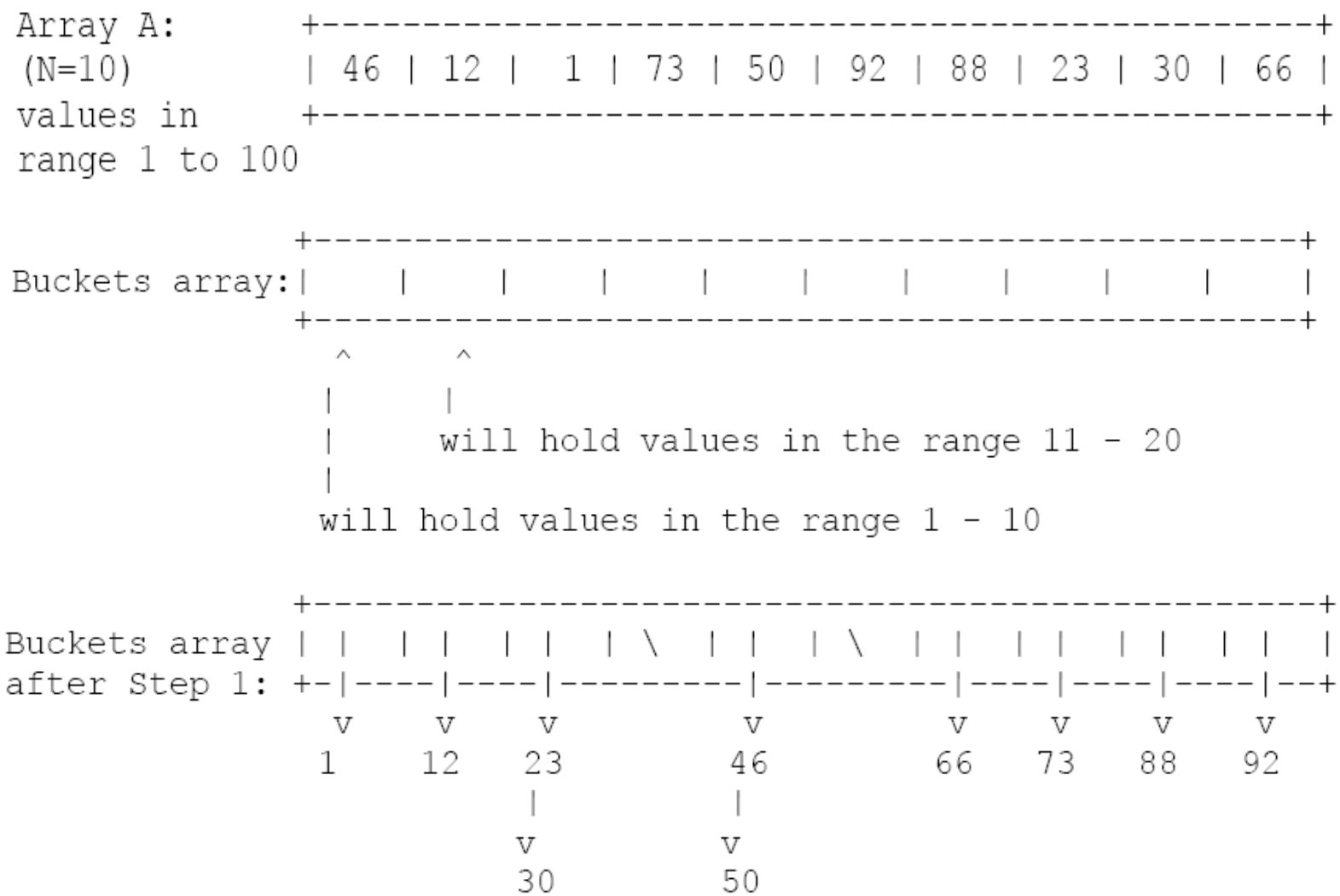
Bucket sort

- How to sort integers using bucket sort?
- The values to be sorted are evenly distributed in some range **min** to **max**
- It is possible to divide the range into N equal parts, each of size k
- Given a value, it is possible to tell in which part of the range it is in

Bucket sort

- Use an array (of size N) of buckets
- The range of values is divided into N equal parts (of size k)
- The first bucket (the first array element) will hold values in the first part of the range (min to min + k -1)
- The second bucket will hold the values in the second part of the range
- Step1:
  - Go through given array of elements once
  - Put each values in its appropriate bucket (maintain each bucket in sorted order)
- Go through buckets B[0] to B[N-1] and put the values back into the original array

Bucket sort


Array A:
(N=10)
values in
range 1 to 100
46 | 12 | 1 | 73 | 50 | 92 | 88 | 23 | 30 | 66
Buckets array:
will hold values in the range 11 - 20
will hold values in the range 1 - 10
Buckets array
after Step 1:
1 12 23 30 46 50 66 73 88 92

Bucket sort

- What is the complexity if the buckets are maintained using linked list?
- What is the time complexity in the best-case scenario?

Data Structures

Bucket sorting

- Stable sorting:
  - Consider a sequence, S = $((k_1, e_1), (k_2, e_2), \ldots, (k_n, e_n))$
  - $(k_i, e_i), (k_j, e_j) \epsilon$ S such that $k_i=k_j$
  - If (k_i, e_i) precedes (k_j, e_j) in S before sorting, the same is the case after soring S
- Stability is important, since applications may want to preserve the initial ordering of the items with the same key

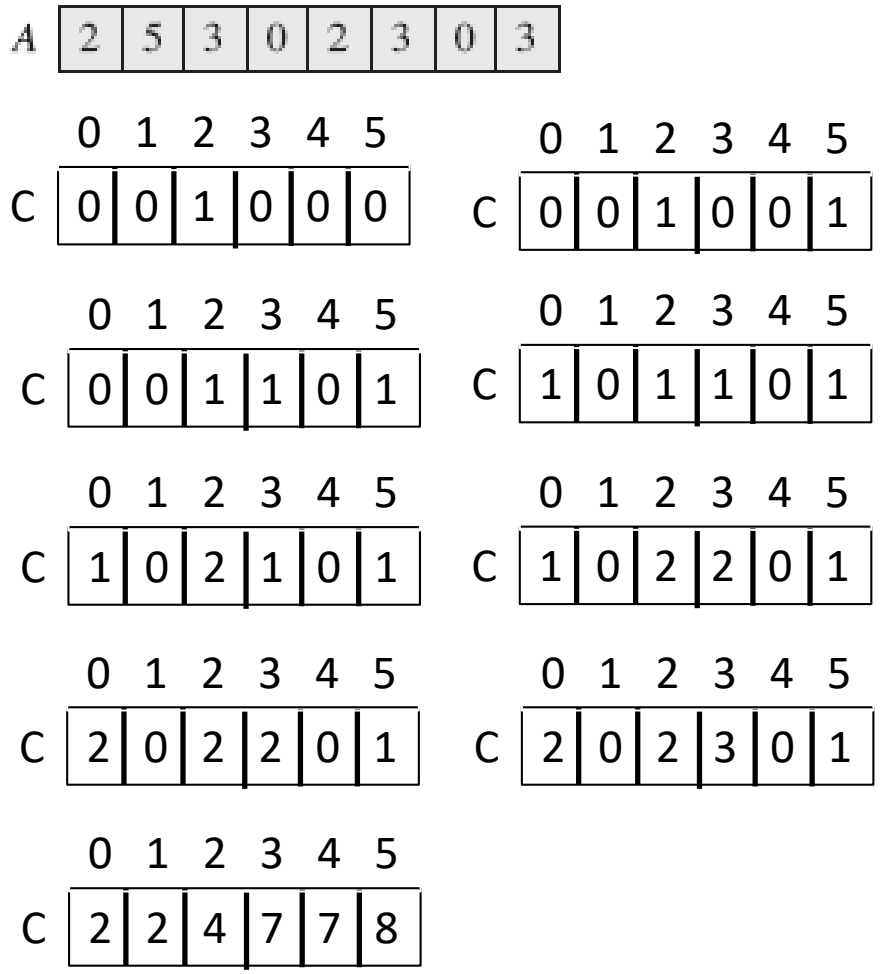
Counting sort

- Assumes that n input elements (integers) are in the range 0 to k, for some integer k
- If k is O(n), the complexity is Θ(n)
- General principle:
  - For each input element x, counts the number of elements that are less than x
  - Based on this information positions x in its correct position
- Has to be modified slightly if input has duplicates

Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

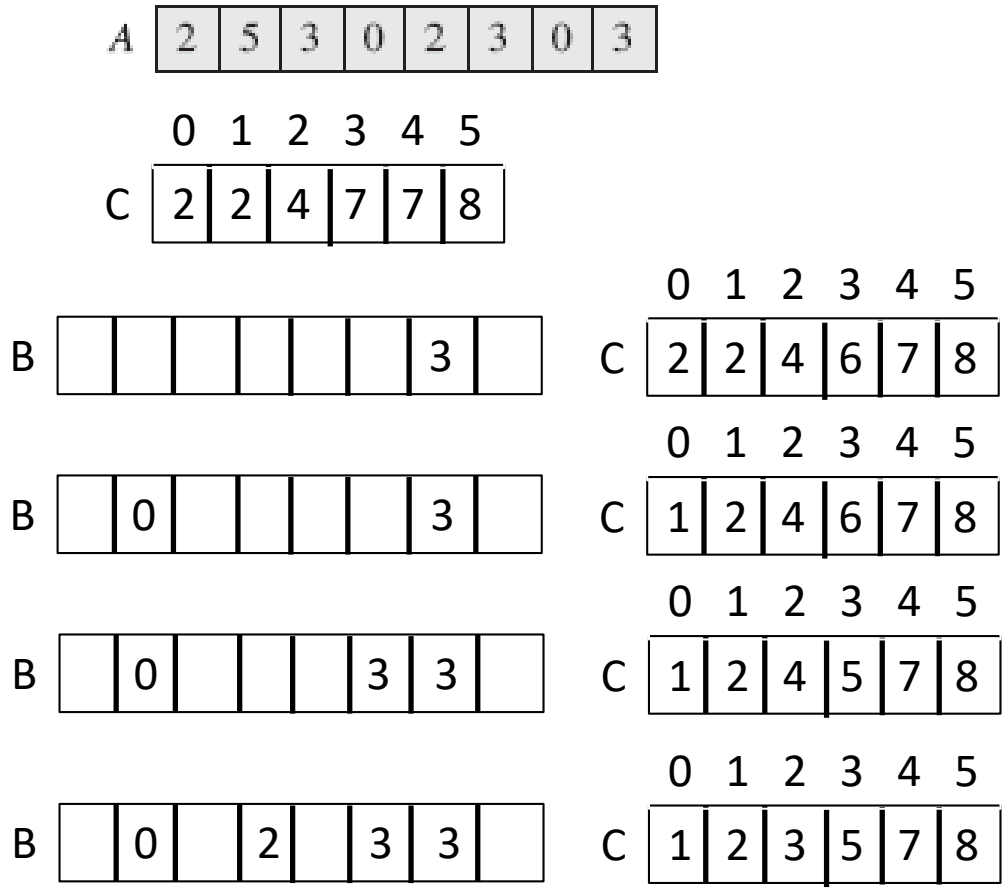
```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

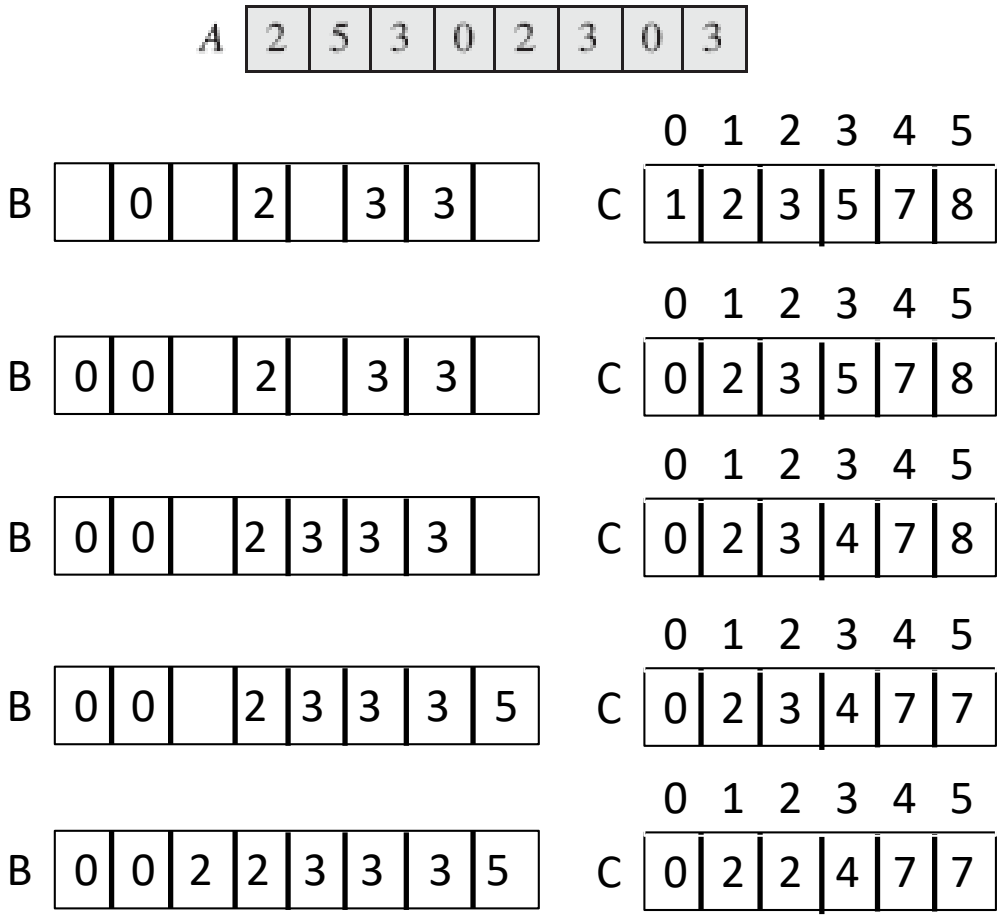
```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}

```
let C[0. . k] be a temporary array
for i ← 0 to k do
        C[i] ← 0
for j ←0 to n-1 do
        C[A[j]] = C[A[j]] +1
for i ← 1 to k
        C[i] = C[i] + C[i-1]
for j ← n-1 to 0
        B[C[A[j]]-1] = A[j]
        C[A[j]] = C[A[j]] -1
```



Counting sort

```
Algorithm Counting_Sort(A, B, k)
{A[0. . n-1] is the input array; B[0. . n-1] holds the sorted output}
        let C[0. . k] be a temporary array
        for i ← 0 to k do
                C[i] ← 0
        for j ←0 to n-1 do
                C[A[j]] = C[A[j]] +1
        for i ← 1 to k
                C[i] = C[i] + C[i-1]
        for j ← n-1 to 0
                B[C[A[j]]-1] = A[j]
                C[A[j]] = C[A[j]] -1
```

Complexity: θ(k) + θ(n) + θ(k) + θ(n) is θ(k+ n)

Radix sort

- How to apply bucket sort in general contexts?
- Consider a sequence of items (k, l), where k and l are integers in the range [0, N-1], for some integer $N \geq 2$
- Lexicographic (dictionary) convention, where $(k_1, l_1) < (k_2, l_2)$ if
  - $K_1 < k_2$ or
  - $K_1 = k_2$ and $l_1 < l_2$
- Can generalize this definition to tuples of d numbers (d > 2)

Radix sort

- Sorts sequence S = ((k_1, e_1), (k_2, e_2), . . ., (k_n, e_n)) by applying bucket sort twice on S
- Using ordering key once and then using the second component
- In which order we should use bucket sort?
- S = ((3, 3) (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))
- S_1 = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))
- $S_{1, 2}$ = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))
- S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))
- $S_{2,1}$ = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))

Radix sort

- First using second component and then by using first component
- If two elements have equal values for second component, then their relative order in the starting sequence is preserved
- We can generalize this sorting to tuples with more than two components

Radix sort

- Consider a sequence of “n” numbers each with at most “d” digits
- Perform bucket sort with respect to the least significant digit
- Continue bucket sort with the next (previous) to least significant digit and end with the most significant digit

Algorithm Radix_Sort(A, d)

for i ← 1 to d

use a stable sorting algorithm to sort A on digit i

Radix sort

064, 008, 216, 512, 027, 729, 000, 001, 343, 125

000, 001, 512, 343, 064, 125, 216, 027, 008, 729

000, 001, 008, 512, 216, 125, 027, 729, 343, 064

000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Radix sort

Algorithm Radix_Sort(A, d)

for i ← 1 to d

use a stable sorting algorithm to sort A on digit i

- Let S be a sequence of n key-element items, each of which has a key $(k_1, k_2, \ldots, k_d)$, where k_i is an integer in the range [0, N-1], for some integer $N \geq 2$; S can be sorted in time O(d(n+N)) using radix sort

Comparison of sorting algorithms

- If implemented well, insertion sort runs in O(n + k) time, where k is the number of inversions in the input
- What is an inversion?
  - A pair of elements that are out of order
- Consider the input: 34, 8, 64, 51, 32, 21
- Inversions: (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), (32, 21)

34	8	64	51	32	21
8	34	64	51	32	21
8	34	64	51	32	21
8	34	51	64	32	21
8	32	34	51	64	21
8	21	32	34	51	64

Comparison of sorting algorithms

- Insertion sort is excellent if the input size is small say less 50
- Effective when number of inversions is small
- $O(n^2)$ (worst-case)
- The merge sort runs in O(n log n) time in the worst-case
- Overhead makes it difficult to implement merge sort in place
- The expected running time of quick sort is O(n log n)
- Worst-case running time is $O(n^2)$
- If we have to sort by integer keys, or d-tuples of integer keys, then we can use bucket or radix sort, where [0, N-1] is the range for keys
- O(d(n+N)) (d =1 for bucket sort)

Data Structures

Dictionaries

- Similar to dictionary of words, facilitates looking up things
- Elements are assigned with keys, keys are used to insert, look up or remove elements
- Dictionary stores items which are key-element pairs (k, e) (key may be the element)
- A symbol table maintained by a compiler
  - Name of an entity is the key
  - Properties of an entity constitute the element part
- Customer records of a bank
  - Account number is the key
  - Other information (personal ad financial) related to a customer is the element part

Dictionaries

- Different data structures to realize dictionaries:
  - Arrays, linked lists
  - Hash tables
  - AVL trees
  - Binary trees
  - B-trees
  - Red/black trees

Dictionaries

- The operations should be supported are:
  - Insertion
  - Removal
  - Searching elements
- Types of dictionaries:
  - Ordered and unordered
  - In both cases key serves as an identifier

Dictionaries

- It can store multiple items with the same key (should not be the case for some applications)
- If keys are unique, then the key is viewed as the address for that item in the memory
- The methods supported by the dictionary ADT are:
  - findElement(k): if there exists an item with key k, returns the element of such an item, and NO_SUCH_KEY otherwise
  - InsertItem((k,e): inserts item (k,e)
  - removeItem(k): if there exists an item with key k, removes such an item and returns the element of the removed item, and NO_SUCH_KEY otherwise
  - NO_SUCH_KEY is known as sentinel

Dictionaries

- Supporting methods:
  - size()
  - isEmpty()
  - elements()
  - keys()
  - findAllElements(k)
  - removeAllElements(k)

Dictionaries

- Consider a telephone service provider
- Has a large number of customers
- How do the company maintain the data of customers so that individual records can be accessed efficiently?
- Data is in the form of (key, customer_details)
- Key is the phone number, which has 10 digits
- n is the number of customers

Dictionary: unordered sequence

- We push the data into the dictionary arbitrarily
- Search operation is O(n)
- Removal is takes O(n)
- Insertion takes O(1)

Dictionary: ordered sequence

- The customer recorded are stored as an ordered sequence (say in the order of keys)
- Search takes O(log n)
- Insertion takes O(n)
- Deletion takes O(n)

Dictionary: direct addressing

- Have an array of 10^{10} cells
- Insert item (k, e) in the cell with index k
- All operations are supported in constant time
- What is the big disadvantage?
- Another issue?

Dictionary: hash table

- Define a table of size n
- Store the item (k, e) in the cell k mod n
- All operations are supported in O(1) time
- Issue?

Hash table

- An efficient way to implement a directory is using hash table
- Worst-case running time is O(n) and expected running time is O(1) (n is the number of items in the dictionary)
- It consists of two major components:
  - a bucket array
  - a hash function
- Bucket array:
  - an array A of size N
  - Each cell of A is thought of as a bucket (container of key-element pairs)
  - N is the capacity
  - An element with key “k” is inserted into bucket A[h(k)]
  - Cells associated with keys not present in the dictionary are holding sentinel, NO_SUCH_KEY (assumption)

Hash table

- Bucket array:
  - If keys are not unique, then we have a “collision”
  - If each cell is capable to hold only one element, then collision is an issue (can be dealt with)
- If the keys are unique and $N \geq n$, then what is the time complexity of operations?
- What about space complexity?
- Define hash table as a combination of a bucket array and a good mapping function from keys to the integers in the range [0, N-1]

Hash table: Hash functions

- The second part of hash table structure is a “hash function (h)” which maps keys to integers in the range [0, N-1], N is the bucket array capacity
- “h” can be applied to arbitrary keys
- h(k) serves as the index into the bucket array, store (k, e) in the bucket A[h(k)]
- A key k should get mapped to the same value each time
- Characteristics of a good hash function: minimize collisions, easy and fast, uniform hashing
- Uniform hashing: each key is equally likely to hash to any of N buckets, independent of where any other key has hashed to

Hash table: Hash functions

- Evaluation of hash function consists of two operations:
  - mapping keys to integers (called hash code)
  - mapping hash code to an integer in the range of bucket array indices (compression map)

Hash table: Hash functions

- First action:
  - Takes an arbitrary key k and maps it to an integer (hash code or value for k)
  - This integer need not be in the range [0, N-1] (could be negative)
  - The hash codes should avoid collisions as much as possible
- Hash codes:
  - cast a long integer representation of a key down to an integer (may lead to a high number of collisions)
  - Sum integer representation of higher order bits and lower order bits
  - This approach can be extended to any kind of key x whose binary representation can be viewed as a k-tuple (x_0, x_1, . . ., x_{k-1}) of integers
  - Character strings can be interpreted as tuple of integers using ASCII character set
  - The hash code of x is: $\sum_{i=0}^{k-1} x_i$

Data Structures

Hash table: Hash functions

- Hash codes:
  - Summation hash code: not a good choice, if keys are either strings or other multiple-length objects viewed as k-tuple (x_0, x_1, . . ., x_{k-1}), where order of x_i's is significant
  - "temp01", "temp10" collide
  - temp01 (116, 101, 109, 112, 48, 49)
  - "spot", "pots", "stop", "tops" collide
  - spot (115, 112, 111, 116)

Hash table: Hash functions

- Hash codes:
  - Integer representation of key x is $(x_0, x_1, \ldots, x_{k-1})$
  - Polynomial hash code: Choose a nonzero constant "a" and use as a hash code the value

    $x_0a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-2}a + x_{k-1}$,

    which can be written as:

    $x_{k-1} + a(x_{k-2} + a(x_{k-3} + \ldots + a(x_2 + a(x_1 + ax_0)). . .))$
  - This hash code uses the components of key "x" as coefficients of the polynomial in "a" (polynomial hash code)
  - Taking "a" to be 33, 37, 39, and 41 produced less than 7 collisions on a list of 50, 000 English words

Hash table: Hash functions

- Second action (compression map):
- Once key object is converted into hash code, it has to be converted into an integer in the range [0, N-1]
- A simple compression map method (division method) is:

$$h(k) = |k| \bmod N$$

- {20, 25, 30, 35, 40, 45, 50}, assume that N =10
- Consider N =11
- If we choose N as a prime, then this method spreads out the distribution of hashed values
- If N is 2^p and then h(k) is p lower-order bits of k

Hash table: Hash functions

- If key values are of the form iN + j for several different i's, then there will be collisions though N is a prime
- The MAD method:
  - Multiply, add, and divide
  - Hash function is defined as: $h(k) = \{ak + b\} \bmod N$; N is prime, a and b are nonnegative integers randomly chosen, $a \bmod N \neq 0$
  - To get close to a good hash function such that the probability of two keys getting hashed to the same bucket is at most 1/N

Hash table: Hash functions

- The multiplication method
  - Multiply key k by a constant A in the range $0 < A < 1$
  - Extract the fractional portion, f, of the result
  - Multiply f by m and take the floor of the result (m is hash table capacity)
  - h(k) = $\lfloor m(kA \ mod \ 1) \rfloor$
  - kA mod 1 = kA - $\lfloor kA \rfloor$
  - Value of m is not critical here
  - Can choose m as a power of 2 (say 2^p) for easy implementation of hash function
  - Assume that the word size of machine is “w” bits and k fits in a word
  - Restrict A to be of form $s/2^w$, where $0 < s < 2^w$
  - Multiply k by w-bit integer $s = A. 2^w$
  - The result is a 2w-bit value $r_1 \ 2^w + r_0$
  - The most significant p bits of r_0 is the hash value of k

Hash table: Hash functions

- Assume that the word size of the machine 8-bits
- Select m = 2^3 and A as 0.25
- Consider key k = 51
- h(k) = $\lfloor m(kA \ mod \ 1) \rfloor$ = $\lfloor 8(0.75) \rfloor$ = 6
- s = A. 2^w = 64
- ks = 51*64 = 3264
- 12*2^8 + 192
- 192: 1100 0000

Collision handling schemes

- Consider two items (k_1, e_1) and (k_2, e_2)
- If $h(k_1) = h(k_2)$, then we have a collision
- Which operations get affected?
  - insertItem() and findItem()
- A simple and efficient way is to have each bucket A[i] to store a reference to an unordered sequence (list), S_i, that stores all items that are mapped to bucket A[i]
- Each bucket is a miniature dictionary
- This way of collision resolution is called separate chaining
- Assume that each nonempty bucket is implemented as a list

Separate chaining

```
Algorithm findElement(k)
        B ← A[h(k)]
        if B is empty then
                return NO_SUCH_KEY
        else
                {search for key in the list for this bucket}
                return B.findElement(k)
```

Separate chaining

Algorithm insertItem(k, e)

    if A[h(k)] is empty then

        Create a new list B, which is initially empty

        A[h(k)] ← B

    else

        B ← A[h(k)]

    B.insertItem(k,e)

Separate chaining

Algorithm removeElement(k)

B ← A[h(k)]

If B is empty then

return NO_SUCH_Key

else

return B.removeElement(k)

Separate chaining

- Insertion runs in constant time (item is not present in the table)
- In the worst-case the time to search an item is $\theta(n)$
- Consider a hash table T of capacity m that stores n items
- Average-case performance depends on how well the hash function distributes keys among m buckets on average
- Assume that any given item is equally likely to hash to any of the m buckets independent of where any other item has hashed to (simple uniform hashing)
- The load factor of T, α, is defined as n/m, that is, average number of elements stored in each list/chain
- α can be less than, equal to, or greater then 1

Separate chaining

- n_j is the length of the list pointed by T[j], where j = 0, 1, . . ., m-1
- $n = n_0 + n_1 + \ldots + n_{m-1}$
- $E[n_j] = \alpha = n/m$
- Assume that the time to compute hash function is O(1)
- Time required to search for an item with key k is linearly dependent on the length $n_{h(k)}$ of the list referred by T[h(k)]
- Analyse the expected number of items examined by the search algorithm, that is, the number of items in the list referred by T[h(k)]
- Consider two cases: unsuccessful search and successful search

Separate chaining

Theorem: In a hash table, if collisions are resolved by chaining, an unsuccessful search takes average-case time $\theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof:

Any key k which is not present in table T is equally likely to hash to any of the m buckets

The expected time to perform an unsuccessful search is the expected time to search to the end of the list of T[h(k)]

The expected length of the list of T[h(k)] is $E[n_{h(k)}]$ is α

The expected number of items examined in an unsuccessful search is α

Total required for an unsuccessful search is $\theta(1 + \alpha)$

Separate chaining

Theorem: In a hash table if collisions are resolved by chaining, a successful search takes average-case time $\theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Proof:

Item to be searched is equally likely to be any of n items stored in the table

The number of items searched in a successful search for an item x is one more than the number of items that precede x in x's list (why?)

Find the number of items that were inserted after x was inserted in x's list

Separate chaining

Proof (contd):

Let x_i be the ith item inserted into the table, for i =1, 2, . . ., n and let $k_i = x_i$.key

For keys k_i and k_j define a random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$

Under the assumption of simple uniform hashing, $Pr\{h(k_i) = h(k_j)\} = 1/m$, so $E[X_{ij}] = 1/m$

The number of items that were searched in a successful search for x_i is: $\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)$

The expected number of items searched in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

Separate chaining

Proof (Contd):

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}1/m\right)$$

$$= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$= 1+\frac{(n-1)}{2m}$$

$$= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Separate chaining

Proof (Contd):

Thus the total time required for a successful search is: $\theta(2 + \alpha/2 - \alpha/2n)$, which is $\theta(1 + \alpha)$

- If hash table capacity is at least proportional to the number of items in the table, then n is O(m)
- $\alpha = n/m$ is O(m)/m, which is O(1)
- Thus searching takes constant time
- If the lists are maintained using doubly linked lists, then removal also takes constant time