CS 234

Data Types and Structures

Spring 2014

Instructor: Kevin Lanctot

Slides based on notes and slides from Lori Case and J.P. Pretti with additions by Kevin Lanctot.

Many diagrams are taken from the course text, Data Structures and Algorithms Using Python by Rance D. Necaise

Table of Contents

Topic	Slide
1. Course Overview	2
2. Python Review	21
3. Abstract Data Types	33
4. Python Lists	64
5. Analysis of Algorithms	72
6. Linked Lists	120
7. The Stack ADT	168
8. The Queue ADT	178
9. The Priority Queue ADT	191
10. Binary Trees	201
11. Sorting	229
12. Searching and Hashing	282
13. Binary Search Trees	334
14. 2-3 Trees	373
15. B-Trees	406
16. Graphs	424
17. Final Exam Notes	478

Welcome to CS 234!
Topic 1 – Course Overview

Four Questions

- Who am I? Who are we? → Staff
- Who are you? → Intended Audience
- What are we doing? → Course Objectives
- How will we do it? → Course Delivery

References

- More details in course syllabus available on Learn.

Who are we? → Staff

Instructor

- Kevin Lanctot, klanctot@uwaterloo.ca
- Office: ~~DC 3136~~ DC 2555C
- Office hours: Tue 2:30-3:30 PM
- Things to know about me:

  I'm a talker not a typer.

  I only check my email once or twice a day.

  Last name is pronounced *long-k toe*, i.e. "*long toe*" with the "*k*" sound after *long*

Who are we? → Staff

Instructional Assistants

- William Saunders, wrsaunde@uwaterloo.ca
- Office: DC 3591 (near front of room)
- Office hours: Monday 2:30-3:30 PM

- Joseph Haraldson, jharalds@uwaterloo.ca
- Office: DC 2302 B
- Office hours: Tuesday 3:30-4:30 PM

- deal with MarkUs, answering questions about course material, issues around assignment marking

Who are we? → Staff

Instructional Support Coordinator

- Ahmed Hajyasien
- email: ahajyasien@uwaterloo.ca
- Office: DC 3115
- pick up assignments in his office, deals with cheating cases

Who are you?

Intended Audience

- Interested in learning more about CS
- Seeking to understand the source of potential improvements and efficiencies in programs
- Have passed CS 116 or CS 136
- Required for CS minor
- CS as a second teachable

What are we doing?

Course Objectives

- To study efficient algorithms and data structures in a language-independent setting.
- To become familiar with a number of standard data structures and algorithm design approaches.
- To gain additional experience in program design and implementation.
- To appreciate the formal analysis of algorithms.
- For example, consider the Jellybean Challenge...

What are we doing?

Lessons Learned from Jellybean Challenge

- Computers are currently very good as solving some problems and very poor at solving others.
- What are we missing from the current state of the art?
- How we store and access data is critical.
  i.e. Data Structures.
- How we use computers solve to problems is critical.
  i.e. Design of Algorithms

What are we doing?

Reality Check

This course is concerned with efficiency. What other aspects of good program design are important?

- use minimal resources (time, space)
- easy to understand, maintain, reuse, extend
- correct, robust, secure
- modular
- scalable
- elegant (i.e. a simple solution to a difficult problem)

How will we do it?

Course Delivery

- Lectures will include

  slides (with some blank sections for discussions)

  some notes made on whiteboard

  technical demonstrations
- Slides only contain key points

  must supplement slides with course text and notes taken in class
- No labs or tutorials

How will we do it?

Assignments

- Written – drop boxes beside door to MC4065
- Programming (Python 2.7.3) – MarkUs
- 5 assignments each due Wednesday at 4 PM.
- 24 hour late, -15% (even if just part of it is late)
- Accommodations for a missed assignment, midterm or final exam required valid Verification of Illness form. Only the current version of the form will be accepted.

How will we do it?

Assignments

If you have a challenging problem to solve, what are some good ways to approach it?

- focus your attention exclusively on solving the problem
- start early, so you have time to think about it
- try solve simpler versions of the problem
- break down the problem into easier subproblems
  - ask what went wrong with your previous approach and try to correct it

How will we do it?

Grade Calculation

- 25% 5 assignments, each worth 5%
- 25% Midterm Exam

  Tuesday June 24, 4:30-5:50 *PM.*
- 50% Final Exam
- ***You must pass the weighted average of the midterm and final exam.*** Otherwise, the maximum course grade is 46%**.**
- Grades (will be) viewable on Learn – check them!

How will we do it?

Some Important Dates

- Assignment 1: Wednesday May 21 at 4:00 PM
- Assignment 2: Wednesday June 4 at 4:00 PM
- Assignment 3: Wednesday June 18 at 4:00 PM
- Midterm: Tuesday June 24, 4:30-5:50 PM
- Assignment 4: Wednesday July 9 at 4:00 PM
- Assignment 5: Wednesday July 23 at 4:00 PM
- Final Exam: to be scheduled by the Registrars' Office

How will we do it?

Course Textbook

- *Data Structures and Algorithms Using Python* by Rance D. Necaise
- Copies will be held on reserve at the DC library
- Cover Chapters 1-9, 11-14, Appendices

Optional Python Language Reference

- *Python Essential Reference,* Fourth Edition, by David M. Beazley
- Particularly Chapters 1-11

How will we do it?

Learn

- lectures slides: do you want pdf? or pptx?

  blank sections in slides updated after lecture
- supplementary material, e.g. sample python code, Jellybean Challenge
- assignments
- assignment solutions
- marks

How will we do it?

Piazza

- For general questions about assignments
- One [official] thread per assignment
- Use private post if hinting about your approach, how you might solve it, implementation details
- When asking about an assignment put the following in your title, AxxxQyyy.

  E.g. if you asking about Question 2 on Assignment 1 include A1Q2 in the heading

How will we do it?

Piazza

- You will receive an invitation to join the class discussion for CS 234 via your uwaterloo.ca email this week.
- For questions about lecture material it is best to see me or one of the Instructional Assistants but you are welcome to use Piazza.
- We will have extra office hours just before the midterm and final exams.

How will we do it?

UW Values

- honesty, trust, fairness, respect and responsibility

Avoid Cheating

- Do your own work.
- Do not try to look up answers to assignment questions on the web (unless the question states that it is ok to do so).
- If you have a weird python error it is OK to check on web what error means.
- Midterm and final are worth 75% of your final mark.

How will we do it?

Academic Integrity
www.uwaterloo.ca/academicintegrity

Avoiding Academic Offences
http://www.math.uwaterloo.ca/navigation/Current/cheating_policy.shtml

Grievance
http://www.adm.uwaterloo.ca/infosec/Policies/policy70.htm

Discipline
http://www.adm.uwaterloo.ca/infosec/Policies/policy71.htm

Appeals
http://www.adm.uwaterloo.ca/infosec/Policies/policy72.htm

Topic 2a – Python Review

Sample Python Scripts

- Python 2 vs. Python 3
- example1.py : simple types, expressions , I/O
- example2.py: functions, control structures
- example3.py: lists, dictionaries, files

References

- Appendices A of course text.
- Supplementary Materials section of Learn

A Tale of Two Pythons

- Python changed between version 2.5 and 3.0
- Version 2.6 and 2.7 are “intermediate versions” that support (some) features of both
- For example

```
print "Hello world!"   # old format of print
print("Hello world!")  # new format of print
```

- Python 2.7.3 supports both
- We’ll always use the new version (forward compatible) if possible.

A Tale of Two Pythons

Python 2.7.3 and 3

- `print ("hi")`
- `input('prompt')`
- `open("in.txt", "r")`
- `a != b or a <> b`
- `9 // 2 => 4`

Python 2.x

- `print "hi"`
- `raw_input('prompt')`
- `file("in.txt", "r")`
- `a != b`
- `9/2 => 4`

Only in Python 3

- `9 / 2 => 4.5`
- `range(n)`

`xrange(n), range(n)`

example1.py

Part 1.1

- no semi-colons to mark the end of a statement
- quotes can be 'single' or "double" for a string
- backslash for special characters
- print evaluates to None but output is always a side effect

example1.py

Part 1.2

- the type() commands tells you want type or class a variable or constant is
- int, float, str, and bool are primitive types
- floats are stored inexactly
- int(), float(), str() convert variables and constants
- the operator '+' means different things to different types
- for strings it means concatenation (i.e. join together)

example1.py

Part 1.3

- precedence rules (when calculating totalPoints)
- no type declarations
- console i/o using input()
- can use a string for the prompts with input()
- long lines can be split with a backslash '\'
- running as a script versus running in the shell
- in shell we still have access to global variables

example2.py

Flow of control

- in `guessNumber(): while ... :`
- in `printDiamond(): for i in range(...):`
- in `num5s(): if ... : elif ... : else:`
- don't forget colons in flow of control structures
- indentation matters

Boolean (bool) expressions

- can contain variables

example2.py

Arguments

- arguments are passed by value
- changing the values of a parameter inside a function does not change its value outside the function

Return types

- no explicit return type `for guessNumber()` and `printDiamond()`
- they return None

example3.py

Lists, Dictionaries and Files

- indexing, slicing and concatenating lists
- empty lists, dictionaries
- tuples do not support mutation (modification)
- many useful methods exist (e.g. append, extend, sort)
- more will come through out the course

Topic 2b – Python Review

Sample Python Scripts

- example4.py : classes
- example5.py: exceptions

References

- Appendices C and D of course text.
- Supplementary Materials section of Learn

example4.py

Classes

- the method __init__ initializes the class
- methods framed with double underscores, like __str__, __eq__, __ne__ represent operators like str(), ==, !=
- see course text pg 501 for more (about 25) operators
- use keyword self to refer to this object
- self typically appears as the first parameter of a method

example5.py

Exceptions

- when python encounters an error it raises an exception, like *ZeroDivisionError*
- you can catch these errors with *try* ... *except*
- you create exceptions with the keyword *raise*
- common exceptions for checking user input are *ValueError* and *TypeError*
- see course text pg 491 for 6 common exceptions
- assert is another type of exception that can be turned off and it typically used to find programming errors

Topic 3 – Abstract Data Types

Key Concepts

- Types, Data Types
- Abstraction
- Abstract Data Types
- Preconditions, Postconditions
- Stock Market ADT example

References

- Chapter 1.1 and 1.2 in course text
- Note: definitions taken directly from the course text

Types and Data Types

Types

- Computers represent data as a sequence of 1's and 0's
- What exactly does the following sequence represent?

  00000001 01011100 10111101 01011100 01110001 01111000 10011111 11111100
- Answer: it depends
- Could be some characters, an integer, a float
- You (or the computer) cannot tell just by looking at it
- Need something to tell us how to interpret the data
- That "something" is ...

Types and Data Types

Data Types

- A *data type* is how to interpret the data.
- A *data type* can be simple
  - Python's *simple data types* include: None, bool , int, float
- A data type can be complex
  - Python's *complex data types* include: str, list, tuple, range, set, dict
- A *complex data type* can be broken up into other data types

Abstraction

Problem: size of computer programs

- some codebases can be *huge*
- Windows 8.1 has roughly 80 millions lines of code
- reading one line of code per second it would take 2.5 years to read it all (with no time off for eating or sleeping)
- for more examples see

  http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

- *Key Question: How do we deal with such complexity?*

Abstraction

Solution: Abstraction

- How does the Python function math.cos(*x*) actually calculate the cosine of the value *x*?
- How does the Python function socket.send(aMsg) actually send a message to another computer?
- *Key Insight: Almost all Python programmers just need to know what these functions do not how they are implemented.*
- *Abstraction means capture the relevant (or important) aspects*

Abstraction

Abstraction in Computer Science

- Focus on ***what*** is done (by a function or with data) rather than ***how*** it is done (the actual implementation details)
- Two types:
  - *Procedural Abstraction*: use a function without knowing the details of how it is written
  - *Data Abstraction*: use data types without knowing the details of how they are implemented

Abstract Data Type

- *An Abstract Data Type* (ADT) is a "programmer-defined data type that specifies a set of data values and a collection of *well-defined* operations that can be performed on those values" (course text pg 3)
- Key Point: an ADT is described through an interface
- independent of a specific programming language
- Implementation Independent: implementation details are hidden (a.k.a. abstracted away) and ADT makes no reference to the actual implementation details
- focus on *what* the ADT can do, not *how* it does it.

Abstract Data Type

Why use Abstract Data Types?

- Focus on basic operations and the use of data instead of low-level implementation details.
- Prevent errors by limiting controlling the use of implementation details (how the type can be used).
- Implementation can be updated without changing programs using the ADT.
- Easier to manage large projects by breaking them down into smaller modules.

Abstract Data Type

Example: the Interface for the Fraction ADT

- Fraction (numerator, denominator): create a fraction from two integers
- Operators {+, -, *}: add, subtract, and multiply two fractions and return a third fraction
- Operator { / }: divides a fraction by a non-zero fraction and returns a fraction
- float(aFraction): converts aFraction to a float
- Operators {==, !=, <, <=, >, >=}: takes two fractions and returns a bool which tells if the first fraction is equal to, not equal to, less than, ... the second fraction

Abstract Data Type

Creating an ADT in Python

- Create a new class
- Describe the data values (types) as a comment
- Describe the operations (the functions of the class)
- For each function, describe preconditions and postconditions (more on this in later slides)
- Do not provide the function body (i.e. implementation)
- The ADT must be fully specified (clear and complete) in terms of the types and operations.

Abstract Data Type

What is Missing from an ADT

- Can't run this code
- Need an implementation of the ADT
  - Follow function descriptions,
  - Assume or check preconditions
  - Guarantee postconditions
  - Include needed fields
  - Add helper functions as needed (functions not mentioned in interface but are helpful in implementing the ADT

Preconditions and Postconditions

Preconditions

- Example: say *aFunction* takes parameters *a* and *b* and returns *c*, i.e. *aFunction*(*a*, *b*) → *c*
- Preconditions
  - conditions the user of *aFunction* has to guarantee are true before using it or ...
  - assumptions the implementer of *aFunction* can make while implementing it
- Users prefer few preconditions, implementers prefer lots of them.

Preconditions and Postconditions

Preconditions

- For example
  - in division with integers, *a* // *b*, a precondition is that *b* is non-zero
  - for math.sqrt(*x*) with floats, a precondition is that *x* is non-negative
- Typical preconditions involve checking ..
  - the type (is it a float) and raising a TypeError exception if it is not
  - the value (is it non-zero) and raising a ValueError exception if it is not

Preconditions and Postconditions

Postconditions

- Example: say *aFunction* takes parameters *a* and *b* and returns *c*, i.e. *aFunction*(*a*, *b*) → *c*
- Postconditions
  - conditions the implementer of *aFunction* has to guarantee after *aFunction* is done (as long as the preconditions have been met).
  - assumptions the user of *aFunction* can make after the function has finished
- Users prefer lost of postconditions , implementers prefer few of them.

Preconditions and Postconditions

Postconditions

- For example: in division with integers, *a* // *b*, the postcondition is that the function will return the integer value of how many times *b* divides *a* ignoring any remainder.

Preconditions vs. Postconditions

- Preconditions usually focus on the inputs to a function
- Postconditions usually focus on the output of a function and any side effect (such as print which has the side effect of printing to the screen or a file)

ADT Example – Stock Exchange

Goal: have 2 programmers create a Stock Exchange app

- Have one programmer implement the stock exchange application using an ADT
- Have another programmer implement the stock exchange ADT
- Both programmer can work at the same time as long as they *agree on the ADT specification*
- i.e. function names, function parameters, preconditions, postconditions

ADT Example – Stock Exchange

Stock Exchange Files

1. Stock.py: a stock data type
2. StockExchangeADT.py: a stock exchange ADT, i.e. just contains the interface no implementation
3. StockExchangeApp.py: a stock exchange application that uses the stock exchange ADT
4. StockExchangeList.py: implements the stock exchange ADT as a list of stocks
5. stock_data.txt: a file that contains stock information

ADT Example – Stock Exchange

Stock Exchange Files

1. Stock.py: lists the components of a Stock, i.e. name, trading symbol, current cost, lowest cost, highest cost, volume
2. StockExchangeADT.py : allows user to add, remove, sell or get the price of a stock (uses the Stock data type)
3. StockExchangeApp.py: asks users which stocks they want to sell and totals the price of all stocks that the user wants to sell (uses the StockExchangeADT)

ADT Example – Stock.py

Stock.py

- defines constructor, string conversion, == and != operators
- __repr__ creates a string representation of the object that captures *all* the information about the object
- %d, %s, %.2f : substitutes integers (%d), strings (%s) and floats with two decimal places (%.2f) into a string

ADT Example – StockExchangeADT.py

StockExchangeADT.py

- contains no implementation (i.e. code) just the specification (function definitions, preconditions and postconditions)
- from Stock import * include the contents of the Stock.py file in this file, making everything local (e.g. math.sqrt() is not local, sqrt() is local)
- pass does absolutely nothing, typically used in ADTs
- __contains__() defines what the *in* operator does, more on *in* later...

ADT Example – StockExchangeApp.py

StockExchangeApp.py

- it allows you to get the value of some stock
- uses methods from StockExchangeADT.py to implement part of a Stock Exchange.
- illustrates *information hiding,* it does not matter how StockExchangeADT.py eventually gets implemented
- exch is the StockExchange object: the methods it must recognize for this app to work are ...

ADT Example – StockExchangeApp.py

StockExchangeApp.py

- StockExchange methods used...

  - exch = StockExchange("stock_data.txt") (in line 5)

  - if stock in exch (in line 27)

  - exch.getPrice(stock) (in line 28)

- Other methods are in StockExchangeADT for Assignment 2

ADT Example – StockExchangeApp.py

StockExchangeApp.py

- The following control structure repeats *someStatements* until *aCondition* is True ...

```
while True:
    someStatements
    if aCondition:
        break
```

- raw_input("Enter Symbol for ...") takes input from the console always interpreted as text (e.g. an input of 12 would be a string not an int)

ADT Example - Implementing an ADT

Questions to Ask

- What is the "best" *data structure*?
  - Physical representation of how data is organized and manipulated
- Can the data be stored “naturally”?
- Can the functionality be achieved?
- Can the operations run in a reasonable amount of time?

ADT Example - Implementing an ADT

Choices for StockExchange

- List of Stock objects
  - Ordered? Unordered?
- Dictionary
  - Keys? Values?
- Which is better? Why?

- Are there other choices?

ADT Example – StockExchangeList.py

__init__() method in StockExchangeList.py

- implements the stock exchange as a list of Stocks
- self._stocks leading '_' means you should not access this variable outside the class (it is "hidden")
- line.split() : coverts a string into a list of words
- data.pop(-1) : removes the last item from the list
- float("12.50") converts string to float
- int("12.50") fails, it is expecting an int in the string
- int(float("12.50")) works, first convert to float, then int

ADT Example – StockExchangeList.py

__init__() method

- except Exception as e: captures any exception and store the description of the exception in the variable e

__contains__() method

- __contains__(): is another standard method in Python, like str() and len(), which implements the *in* method typically used in for loops, e.g.
  for x in StockExchange:
  or in if statements, e.g.
  if x in StockExchange:

ADT Example – StockExchangeList.py

_findStock() method

- helper method not specified in ADT specification
- leading underscore '_' means "private"

  - for use only by other methods in this class,

  - not for use outside this class

biggestGainers() method

- variables and function parameters can be ints, floats, strings etc. *and* they can also be *functions*
- sort() can take a function as a parameter (i.e. how to sort? based on what value?)

ADT Example – StockExchangeList.py

biggestGainers() method: functions as parameters

- use lambda to create a function without a name
- lambda a, b: a+ b is a short form for the following...

```
def aFunction(a, b):
    return a + b
```

- A function defined by lambda is called an *anonymous function* (it has no name) and it is usually used when you want to pass a simple function as a parameter to another function, like sort()

example7.py

Testing, Types and Exceptions

- Can test types for TypeError

```
from types import IntType, FloatType
if type(aNumerator) != IntType:
```

- Can include a message with an exception

```
raise ValueError, "aDenominator is 0"
```

- Can conditionally run statements when used as script but not when included as a file

```
if __name__ == "__main__":
```

example8.py

String Operations

- Can split a large string into a list of lines.

```
lines = msg.splitlines()
```

- Can split a line into a list of words

```
lines[0].split()
```

Topic 4 – Python Lists

Key Concepts

- Python Lists
- What's Really Happening
- List Commands

References

- Chapter 2 in course text, especially 2.1 and 2.2
- Note: diagrams taken directly from the course text

Python Lists

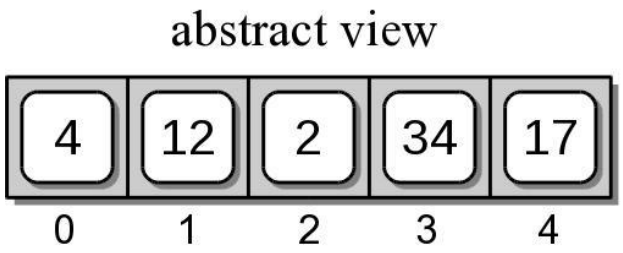
How are Python lists actually implemented?

- 1-d arrays
- Implemented at the hardware level
- Linear collection, with single name
- Contiguous block of memory
- Random access for retrieval and updates
- *Size fixed when created*
- *(Often) Store only one type of data*

Python Lists: Construction

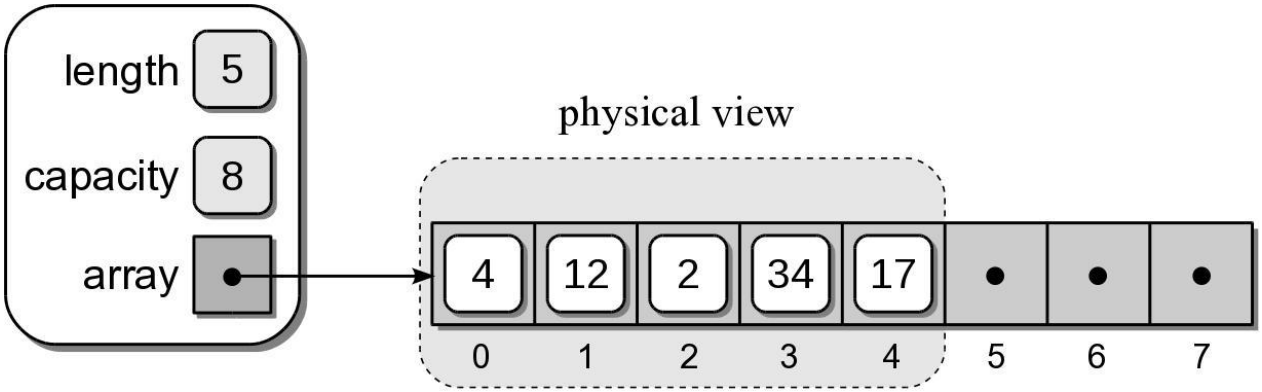
- The Python list interface provides an *abstraction* to the actual underlying implementation.
- Actually a hybrid (a combination) of
  - a list (slow to find an item, fast to add an item) and
  - an array (fast to find an item, slow to add an item)

```
pyList = [ 4, 12, 2, 34, 17 ]
```



Python Lists: What is really happening?

- An array is used to store the items of the list.
  - Created bigger than needed.
  - Has capacity for future items.
  - **subarray** – the items are stored in a contiguous subset of the array.

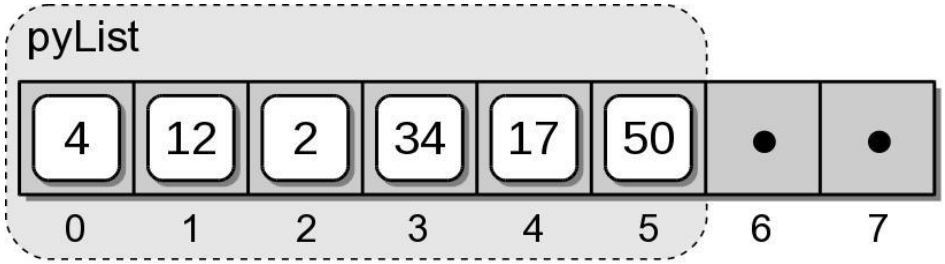


Python Lists: Appending Items

- New items can be added at the end of the list.

  `pyList.append(50)`

- When space is available, the item is stored in the next slot.

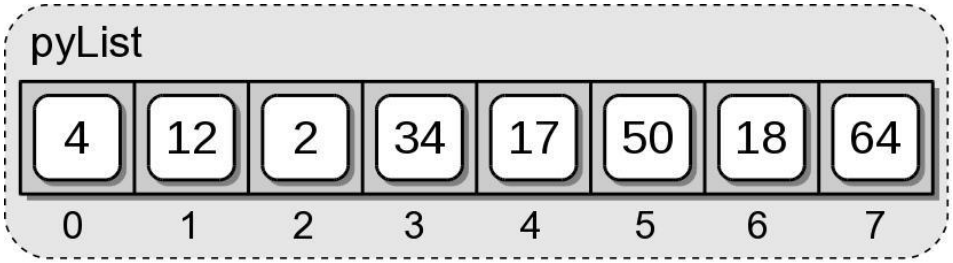


Python Lists: Appending items

- What happens when the array becomes full?

```
pyList.append(18)
pyLisy.append(64)
pyList.append(6)
```

- There is no space for value 6.



- What now ?

Python Lists: What happens for these operations?

- Extending a list

```
pyListA = [ 34, 12]
pyListB = [4, 6, 31, 9]
pyListA.extend( pyListB )
```

- Inserting items

```
pyList.insert(3, 79)
```

- Removing items

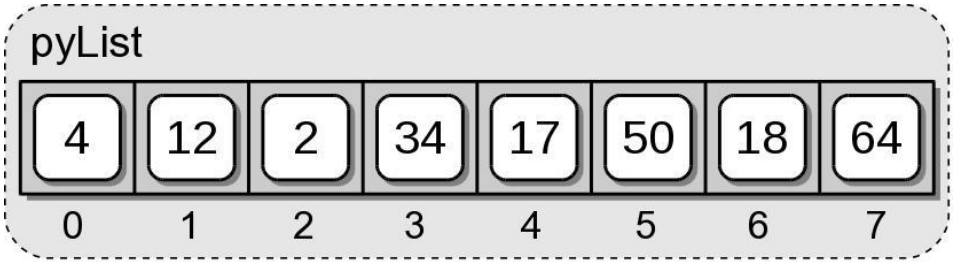
```
pyList.pop()
pyList.pop(2)
```

Python Lists: Other List Operations

- Copying a portion of a list

```
aSlice = pyList[2:5]
```

- Lists start at 0 and end at len(pyList) -1



pyList[0] = 4

pyList[7] = 64

len(pyList) = 8

Topic 5 – Analysis of Algorithms

Key Concepts

- Asymptotic Complexity and Big-O Notation
- Other Measures of Efficiency: Ω and Θ
- The Complexity of Lists and Loops
- Big-O Properties
- Amortized Analysis

References

- Chapter 4.1, 4.2, 4.3 in course text

Complexity

How to measure efficiency?

- run time: implement and run the program
  - influenced by the computer (processor speed and memory), operating system, background processes, the programmer, the data, etc.
  - units of time: msec, sec, minutes, hours, days
- complexity: analyze the algorithm
  - a mathematical approach
  - derive a function of the size of the problem, n
  - sets of functions: $O(\log n)$, $O(n)$, $O(n \log n)$, $0(n^2)$

Complexity

Analyzing Complexity

- *n* is the size of the input, could be in
  - bytes, number of characters in a string, or number of items in input
- T(*n*) could characterize the number of steps, number of comparisons, number of floating point operations (flops) etc.
- We analyze by deriving a formula for *T(n)*
- Two common measurements:
  - Worst case
  - Average case

Asymptotic Complexity: Big-*O* Notation

Positive Integers

- The infinite set { 1, 2, 3, 4, ...} can be represented by either of several symbols
  - $\mathbb{N}^+$ or $\mathbb{N}^*$ representing the *natural number* or
  - $\mathbb{Z}^+$ representing the *non-negative integers*
  - typically use n and m for variables (not x, y, or z)

The Set Big-*O*

- Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $O(f(n))$ if there exist constants $c > 0$ and $m > 0$ such that $T(n) \leq cf(n)$ for all $n \geq m$.

Asymptotic Complexity: Big-*O* Notation

The Set Big-*O*

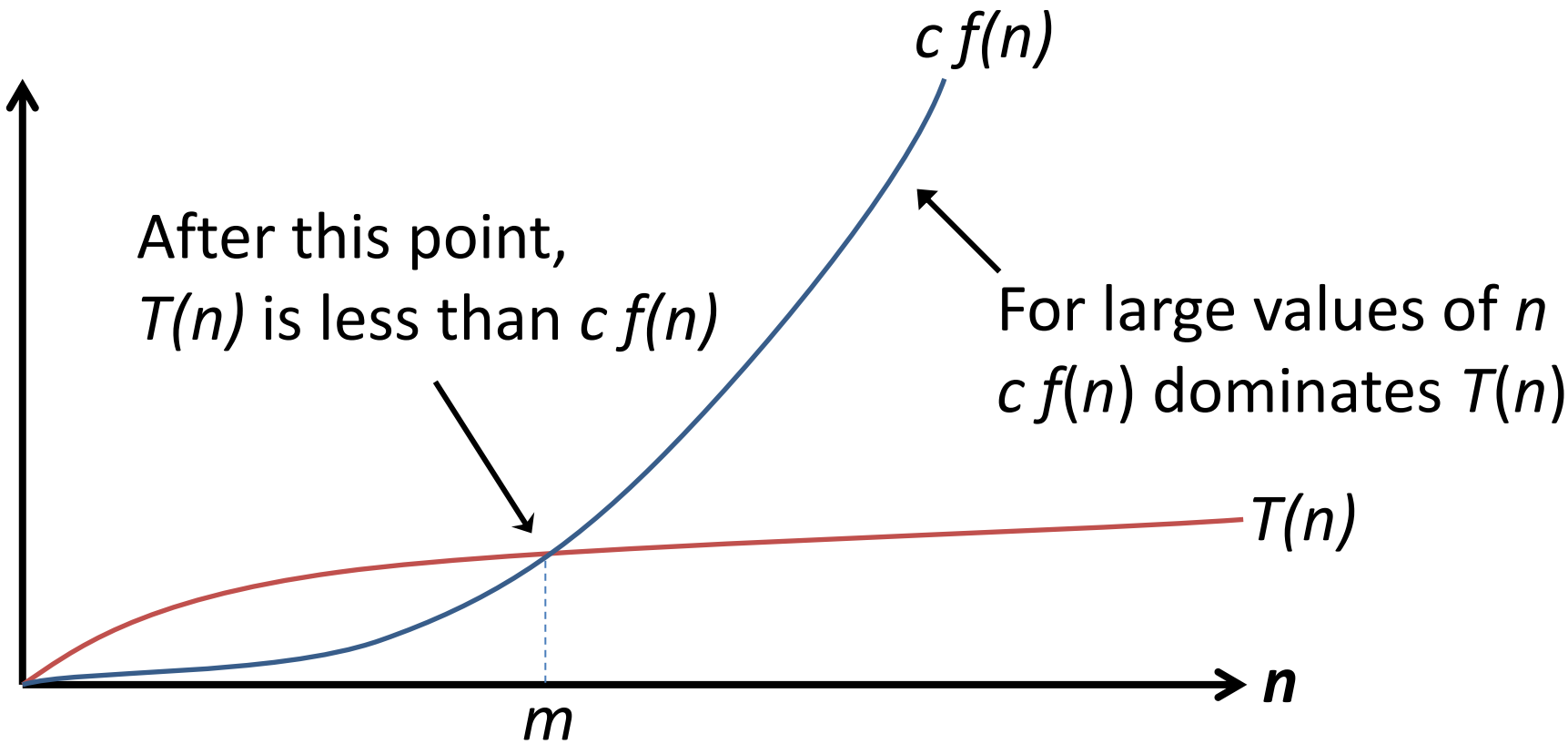
In English, given two functions that have positive domains and ranges, $T(n)$ and $f(n)$

- $T(n)$ is *in the set* $O(f(n))$
- if beyond some point (the point m)
- the function $T(n)$ is dominated by $cf(n)$

Asymptotic complexity is not concerned with small input sizes (i.e. when n is small)

Asymptotic complexity is concerned with large input sizes (i.e. when n is greater than m) *independent of any implementation*

Asymptotic Complexity: Big-*O* Notation



Asymptotic Complexity: Big-*O* Notation

The Set Big-*O*

- Show that $T(n) = ½\, n^2 + 4n + 16$ is in $O(n^2)$.
- $T(n) = ½\, n^2 + 4n + 16$

  $\leq ½\, n^2 + 4n^2 + 16n^2$ (for n ≥ 1)

  $= 20\, ½\, n^2$

  $\leq 21\, n^2$
- Pick $c = 21$, i.e. $f(n) = 21\, n^2$
- For $n = 1$, T(1) = 20 ½ and $f(1) = 21$, hence

  $T(n) = ½\, n^2 + 4n + 16$

  $\leq 21\, n^2 = f(n)$, for n ≥ 1.

Asymptotic Complexity: Big-*O* Notation

The Set Big-*O*

- Show that $T(n) = ½\, n^2 + 4n + 16$ is in $O(n^2)$.
- $T(n) \leq 21\, n^2 = f(n)$, for $n \geq 1$.

Note

- The first derivative of $f(n) - T(n)$ is positive for $n > 0$
- So this function continues to increase in value
- i.e. $f(n)$ is always bigger than $T(n)$
- $T(n)$ is in $O(n^2)$ because there exist constants *21* and 1, such that $T(n) \leq 21\, n^2$ for all $n \geq 1$.

Asymptotic Complexity: Big-*O* Notation

- The function $f(n)$ indicates the *rate of growth* of how the complexity increases as the input size increases
- Answers the question:
  - If we increase the size of the input, how much longer does it take to solve the problem?
  - Most algorithm finish quickly if the input is small, not really worried about this case
- $T(n)$ is bounded above by a multiple of $f(n)$
- $T(n)$ is in the $O(n^2)$ family of algorithms

Asymptotic Complexity: Big-*O* Notation

Functions grow more quickly

T(n)	Common Name
O(1)	Constant
O(log *n*)	Logarithmic
O(*n*)	Linear
O(*n* log *n*)	Log Linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k), k > 0	Polynomial
O(a^n), a > 1	Exponential

Algorithms run more slowly

Asymptotic Complexity: Big-*O* Notation

n	1	log n	n	n log n	n^2	n^3	2^n
1	1	0	1	0	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65,536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,096	262,144	18,446,744,073, 709,551,616

18,446,744,073,709,551,616 seconds
= 584,942,417,355 years.

Asymptotic Complexity: Example

Example: Define and classify *T(n)*

```
def adjust(L, p):
    pos = p
    while pos < len(L):
        L[pos] = -L[pos]
        pos = pos + p

import random
# assume n and pr have been initialized
L = [0] * n
for i in range(n):
    L[i] = random.randint(0,100)
adjust(L, pr)
```

Asymptotic Complexity: Example

Example: Derive the complexity T(*n*)

- Initialization of L takes *n* steps
- range(*n*) takes *n* steps
- for i in range(n): takes *n* steps
- Random.randint
  - independent of *n*
  - say some constant *k* take *k* steps,
- makes a call to adjust()

Asymptotic Complexity: Example

Example: Derive the complexity T(*n*) continued...

- adjust takes 1 step for pos
- each iteration of the while loop takes 2 steps
  - update L[pos]
  - update pos

- How many iterations of while are there? n/p
- Multiplied together, adjust takes $2n/p$ steps

Asymptotic Complexity: Example

Example: Derive the complexity T(*n*) continued...

Added all together, we have

$T(n) = 3n + k + 1 + 2(n/p)$ [note: $n/p \leq n$ for p in $\mathbb{Z}^+$]

$\leq 5n + k + 1$

$\leq 6n$ for all $n \geq k + 1$

Therefore $T(n)$ is in $O(n)$.

Considerations when calculating *T(n)*

- Interested in "dominant" term
- For example, if $T(n) = \frac{1}{2} n^2 + 4n + 16$
- Interested in n^2, $O(T(n)) = \cancel{\frac{1}{2}} n^2 + \cancel{4n} + \cancel{16}$
- Group together similar expressions (e.g. constants, products of n, n^2, etc.)
- For a loop:
  - Determine number of iterations
  - Determine running time of each iteration independently
  - Add them all together

Asymptotic Complexity

Are constants important?

- Yes, when comparing two functions with a similar dominant term and n is small, $T(n) = 20n$ vs. $T(n) = 15n$
- In the C++ library the default sort will use
  - a $O(n^2)$ algorithm (with a small constant) when there are less than or equal to 20 items and
  - a $O(n \log n)$ algorithm (with a larger constant) when there are more than 20 items and
  - If $T(n) = 5\, n \log n$, then $T(20) = 432$
  - If $T(n) = 1\, n^2$ then $T(20) = 400$ ← faster

Other Asymptotic Measures

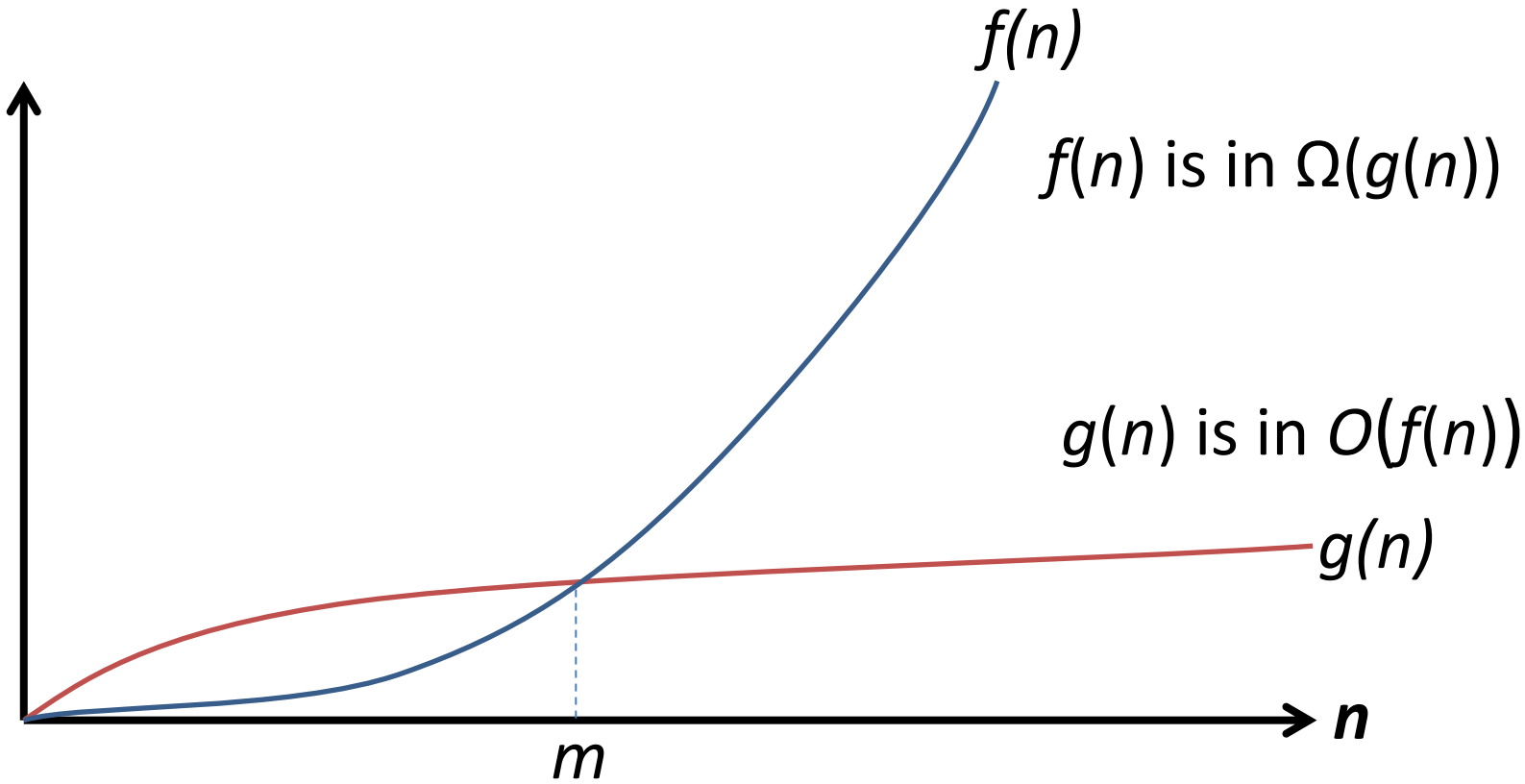
Lower Bounds

- Suppose $T(\mathrm{n})$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$
- We say that $T(n)$ is in $\Omega(f(n))$ if $f(n)$ is in $O(T(n))$

- $T(n)$ is *bounded from below* by $f(n)$.
- Informally, $T(n)$ *grows* at least as fast as $f(n)$.
- More formally...

  Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $\Omega(f(n))$ if there exist constants $c > 0$ and $m > 0$ such that $cf(n) \leq T(n)$ for all $n \geq m$.

Other Asymptotic Measures



- Informally, if $g(n)$ is a *lower bound* of $f(n)$ then $f(n)$ is an *upper bound* of $g(n)$.

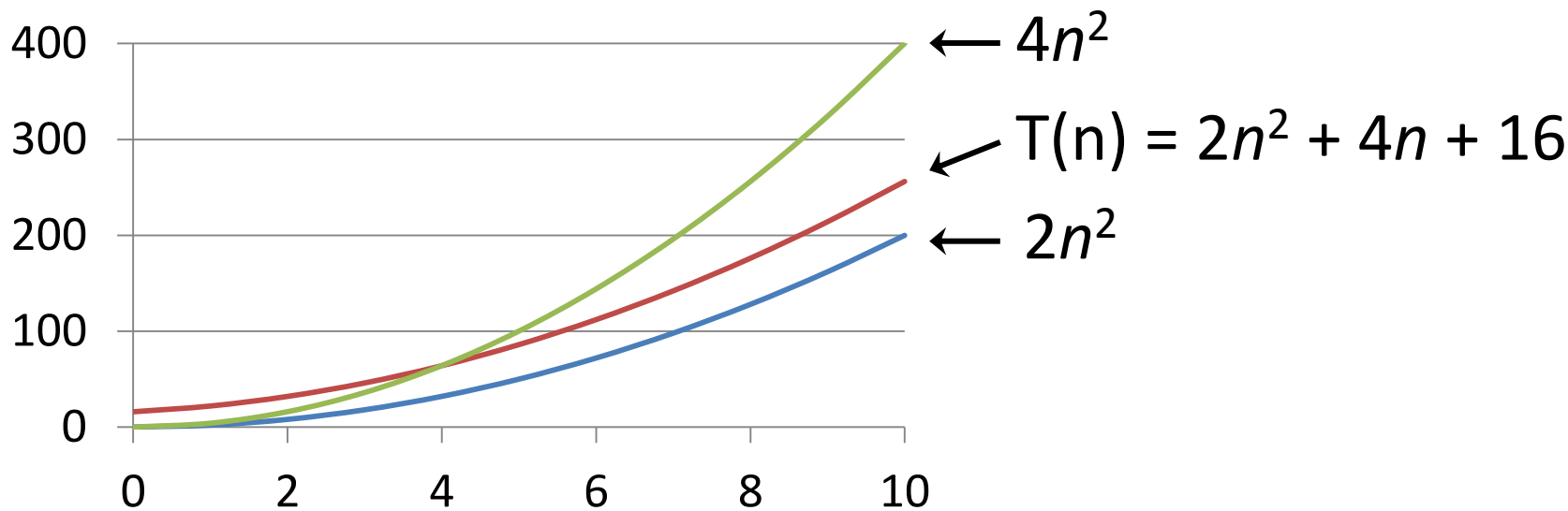
Other Asymptotic Measures

Tight Bounds

- Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$
- We say that $T(n)$ is in $\Theta(f(n))$ if $f(n)$ is in $O(T(n))$ and $T(n)$ is in $O(f(n))$.
- $f(n)$ provides a tight bound on T(n)
- Informally, $f(n)$ and $T(n)$ *grow at the same rate.*
- More formally...

  Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $\Theta(f(n))$ if there exist constants c_1, $c_2 > 0$ and $m > 0$ such that $c_1 f(n) \leq T(n) \leq c_2 f(n)$ for all $n \geq m$.

Other Asymptotic Measures



- When $n \geq 5$, the function $T(n) = 2n^2 + 4n + 16$ is bounded between $2n^2$ and $4n^2$ so T(n) is in $\Theta(n^2)$.
- $T(n) = 2n^2 + 4n + 16$ is in $\Theta(n^2)$ since there exist constants $2, 4 > 0$ and $5 > 0$ such that $2n^2 \leq T(n) \leq 4n^2$ for all $n \geq 5$.

Other Asymptotic Measures

Common Elements in Definitions

- All 3 definitions have much in common:
  - $T(n)$: the *time complexity* of the algorithm we are interested in
  - $f(n)$: the "*name*" of the family of functions that $T(n)$ belongs to, e.g, n, $n \log n$, n^2, n^3
  - c, c_1, c_2: the *constants of proportionality* that multiply the "name" of the family
  - m: a *point* beyond which all the *smaller terms* in $T(n)$ *no longer matter*

Other Asymptotic Measures

Different Elements in Definitions

- All the differs in the three definitions is the relationship between $f(n)$ and $T(n)$

$\Omega(f(n))$	*lower bound*	$cf(n) \leq T(n)$
$O(f(n))$	*upper bound*	$T(n) \leq cf(n)$
$\Theta(f(n))$	*tight bound*	$c_1 f(n) \leq T(n) \leq c_2 f(n)$

Other Asymptotic Measures

Comparing Formal Definitions

- Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $\Omega(f(n))$ if there exist constants $c > 0$ and $m > 0$ such that $cf(n) \leq T(n)$ for all $n \geq m$.
- Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $O(f(n))$ if there exist constants $c > 0$ and $m > 0$ such that $T(n) \leq cf(n)$ for all $n \geq m$.
- Suppose $T(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$ and $f(n)$: $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$, then $T(n)$ is in $\Theta(f(n))$ if there exist constants c_1, $c_2 > 0$ and $m > 0$ such that $c_1 f(n) \leq T(n) \leq c_2 f(n)$ for all $n \geq m$.

Back to Algorithms

Time Complexity of Algorithms

- Let algorithm A be an algorithm with a worst case time complexity described by the function $T(n)$.
- If $T(n)$ is in $O(f(n))$ for some function $f(n)$, we say that A is in $O(f(n))$ or that A has a worst case time complexity in $O(f(n))$

Time Complexity of Some List Functions

Function	Best-case	Worst-case
`L = []`	O(1)	O(1)
`L[i]`	O(1)	O(1)
`len(L)`	O(1)	O(1)
`L + L`	O(n)	O(n)
`L[1:]`	O(n)	O(n)
`range(n)`	O(n)	O(n)

Assume that the length of the list L is n.

Time Complexity of Some List Functions

Function	Best-case	Worst-case
`copy.deepcopy(L)`	O(n)	O(n)
`L.reverse()`	O(n)	O(n)
`L.append(x)`	O(1)	O(n)
`L.extend(L)`	O(n)	O(n)
`L.pop(ind,x)`	O(1)	O(n)
`L.remove(x)`	O(n)	O(n)

Assume that the length of the list L is n.

Time Complexity of Loops: Example 1

```
def oddSumOfProducts1(L):
    sum = 0
    for k in range(1, len(L), 2):
        sum += k*L[k]
    return sum
```

range starts at 1 and goes to the length of the list L with a step size 2, i.e. 1, 3, 5, 7, ... up to len(L).

sum = 0 is O(1), k in range loop repeats $n/2$ times and does O(1) work each time: T(n) is in union of O(1) and O(n) which is O(n)

Time Complexity of Loops: Example 2

```
def sumOfProducts2(n):
    sum = 0
    for k in range(n/2,4*n,8):
        q = 1
        while q < n:
            sum += k*q
            q = q * 2
    return sum
```

sum = 0 is in O(1), k in range loop repeats ($n/2 - n/16$) which is O(n) times, while q < n loop repeats log n times and does O(1) work each time: T(n) is in union of O(1) and O(n log n) which is O(n log n).

Time Complexity of Loops: Example 2

More Details

Q: How many times can you double the number 1 before I get a number larger than n?

A: $\log_2 n$

In the analysis of computer algorithms log n usually means $\log_2 n$ not the more common $\log_{10} n$.

If an outer loop runs *n* times, I must sum what the inner loop does over *n*: easy if the inner loop does constant amount of work.

Time Complexity of Loops: Example 3

```
def NestedLoops(n):
    sum = 0.0
    for i in range(1, n):
        for j in range(1, i):
            sum = sum + (i-j)**2
            sum = sum / i
    return sum
```

The number of iterations the inner loop does depends on the value of *i* which changes each iteration of the outer loop. Number of iterations = $\sum_{i=1..n} O(i) = O(n^2)$

Time Complexity of Common Types

- operations on a single float, int, bool or a string of fixed size are O(1) e.g.

```
price = float(data[-10])
```

- simple assignment of strings is O(1), e.g.

```
self.stockName = name
```

- dealing with strings and files is similar to dealing with lists: any operating that in the worst case requires going throught the whole list, file or string is O(n), e.g.

```
line = aFile.readlines()
```

- more complicated operations can have larger complexities, e.g. merge sort is O(n log n)

Big-O Properties

Repeated Code

- $cf(n)$ is $O(f(n))$ for all $c > 0$

  i.e.repeating a piece of code, f, a constant number of times, c, does not change the asymptotic time complexity

- $O(f(n) * g(n))$ is $O(f(n)) * O(g(n))$

  repeating a piece of code, f, a non-constant number of times, g, changes the asymptotic time complexity

Big-O Properties

Pieces of Code

- $O(f(n) + g(n)) = O(\max(f(n), g(n)))$,

  i.e. following one piece of code, f, by another, g, is the max of each piece of code

- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$

  transitivity, the bound of one piece of code can be used to determine the bound of another piece of code

Big-O Properties

Polynomials and Logs

- n^k is $O(n^p)$ for all $p \geq k$
- polynomial time algorithms are bounded from above by higher order polynomials

- $\log_a n$ is $O(\log_b n)$ for all $a, b \geq 2$
- but logs of any base (a or b) have the same asymptotic complexity

Comparing Algorithms

The Task

Given an list of *n* numbers, find the slice with the maximum non-negative sum, i.e. maximize:

$$L[j] + L[j+1] + \ldots + L[k] \text{ for integers } 0 \le j \le k \le n-1.$$

Example (*i*):

5, 10, -50, 116, 234, -60, 10, 4,

Example (*ii*):

5, 10, -10, 116, 234, -10, 10, 4

Comparing Algorithms

Two Algorithms

- A "brute force" $\Theta(n^3)$ algorithm
- A "clever" $\Theta(n)$ algorithm

Ask the question: Is *asymptotic time complexity* actually relevant when compared to running on a *faster computer*.

Source: Jon Bentley, "Programming Pearls: Algorithm Design Techniques", *Comm. ACM 27*, 9 (Sept. 1984), pp 865-871

Comparing Algorithms

The "Brute Force" Algorithm

```
# try all possible combinations of i, j
def max_sum_slice_v1(seq):
    n = len(seq)
    max_sum = 0
    for i in range(n):
        for j in range(i,n):
            this_sum = sum(seq[i:j])
            if this_sum > max_sum:
                max_sum = this_sum
    return max_sum
```

Comparing Algorithms

Analysis of "Brute Force" Algorithm

This version does less work

```
for i in range(n/3):
    for j in range(2n/3,n):
        this_sum = sum(seq[i:j])
```

outer loop is	$^1/_3\, n$
inner loop is	$^1/_3\, n$
sum is at at least	$^1/_3\, n$
over all it is at least	$^1/_{27} n^3$ which is $\Omega(n^3)$

Comparing Algorithms

The "Clever" Algorithm

```
def max_sum_slice_v2(seq):
    maxSum = 0    # the eventual answer
    maxHere = 0   # best answer so far
    for s in seq:
        maxHere = max(maxHere + s, 0)
        maxSum = max(maxSum, maxHere)
  return maxSum
```

The "`for s in seq`" is $\Theta(n)$ everything else is $\Theta(1)$.

Comparing Algorithms

Understanding the Algorithm

Key Point: as long as you have a positive sum (*maxHere*) keep it, otherwise start over.

Example (*i):*

```
seq     = 5, 10, -50, 116, 234, -60,  10,   4
maxHere = 5, 15,   0, 116, 350, 290, 300, 304
maxSum  = 5, 15,  15, 116, 350, 350, 350, 350
```

Example (*ii):*

```
seq     = 5, 10, -10, 116, 234, -10,  10,   4
maxHere = 5, 15,   5, 121, 355, 345, 355, 359
maxSum  = 5, 15,   5, 121, 355, 345, 355, 359
```

Comparing Algorithms

Bentley's Implementations

- Compare Brute Force on a *slow compute*r to Clever on a *fast computer*
- Brute Force $\Theta(n^3)$ Algorithm implementation on *fast* (for its time) computer $3.0*n^3$ nanoseconds
- Clever $\Theta(n)$ Algorithm implementation on a *slow* (for its time) computer $19.5*n$ milliseconds

Comparing Algorithms

n	Brute Force Fast impl.	Clever Slow impl.
10	0.000003 s	.195 s
100	0.003 s	1.95 s
1000	3 s	19.5 s
2500	47 s	49 s
10000	50 min	3.25 min
100000	34.7 days	32.5 min
1000000	95 years	5.4 hours

Amortized Analysis

Special Case: List Append

Consider the loop:

```
L = []
for j in range(1, n+1):
    L.append(j)
```

What is the worst-case asymptotic time complexity?

How often does the worst-case for **`append`** actually happen for this sequence of steps?

Amortized Analysis

01

1 copied

01 02

2 copied

01 02 03 04

4 copied

01 02 03 04 05 06 07 08

8 copied

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16

+ 16 copied

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

total = 31 copies

Amortized Analysis

How to calculate the Amortize Cost

Given a sequence of n operations,

1. compute the *total cost* over the entire sequence: $T(n)$
2. then compute the average cost per operation: $T(n)/n$

- For some algorithms, *most* operations are very *fast* and only a few are very slow.
- Don't just consider the worst case, but also consider how often it occurs.

This type of analysis is called *Amortized Cost.*

Amortized Analysis

List Append for *n* = 32

- Assume the underlying array has initial capacity 1.
- How many copy statements are needed to create the final list?
- Consider the input size $n = 32$,
- It takes $T(n) = 1 + 2 + 4 + 8 + 16 = 31$ copies
- $T(n)/n = 31 / 32 < 1.$
- The *amortized cost* to grow the list is O(1).

Amortized Analysis

List Append for $n = 2^k$

- Assume the underlying array has initial capacity 1.
- How many copy statements are needed to create the final list?
- Input size $n = 2^k$,
- It takes $T(n) = 1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$ copies
- $T(n)/n = (2^k - 1) / (2^k) < 1$.
- The *amortized cost* to grow the list is O(1).
- *Amortized cost* is not the same *Average cost.*

Topic 6 – Linked Lists

Key Concepts

- The Problems with Arrays
- Solution: Linked Lists
- The Sequence ADT
- Implementing the Sequence ADT with Linked Lists

References

- Linked Structures, Sections 6.1, 6.2, 6.4 in course text
- Advanced Linked Lists, Sections 9.1, 9.2 in course text

The Problems with Arrays

Inserting Data is Expensive

- To *insert* data in the middle of the array, data in the rest of the array must be copied over into new locations: *expensive!*
- E.g. insert the value 46 between 45 and 55

```
17 42 31 45 55 87 88 19 21 63 41 11 ...

17 42 31 45    55 87 88 19 21 63 41 11 ...

17 42 31 45 46 55 87 88 19 21 63 41 11 14
```

The Problems with Arrays

Deleting Data is Expensive

- To *delete* data in the middle of the array, data in the rest of the array must be copied over into new locations: *expensive!*
- E.g. delete the value 88

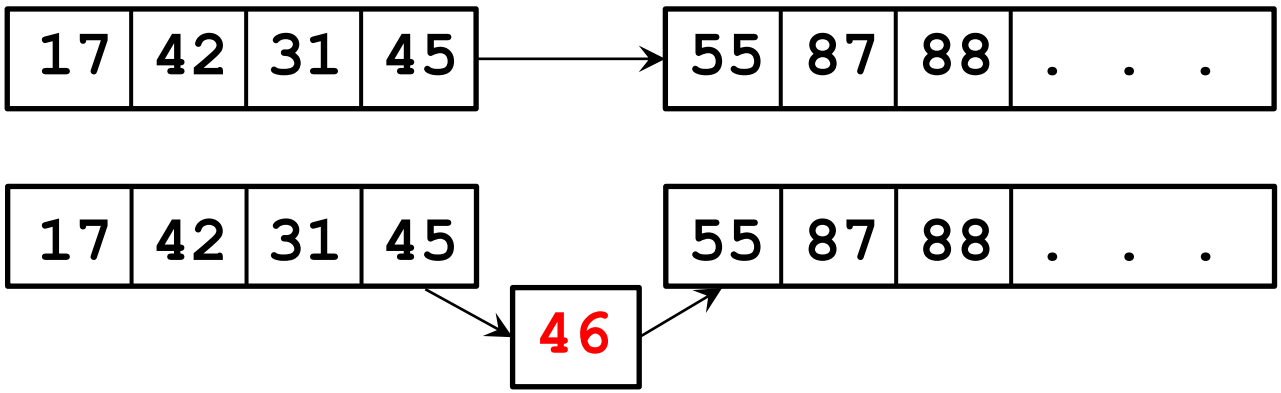
```
17 42 31 45 46 55 87 88 19 21 63 41 11 ...

17 42 31 45 46 55 87    19 21 63 41 11 ...

17 42 31 45 46 55 87 19 21 63 41 11 ...
```

The Problems with Arrays

Idea for Reducing Costs

To insert some data, it would be nice to ...

- have the *first part* of the array *point to* the *new value*
- and have the new value point to the *rest of the array*
- E.g. insert the value 46 between 45 and 55



Linked Lists

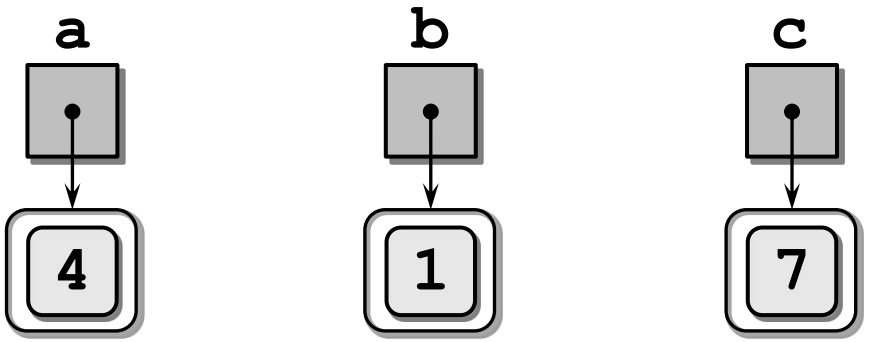
Step 1: Store the Data

- create a class, Node, to store the data

```
class Node:
    def __init__(self, data):
        self.data = data
```

- create 3 objects: Node(4), Node(1), Node(7)
- also create 3 variables: a, b, c

```
a = Node(4)
b = Node(1)
c = Node(7)
```

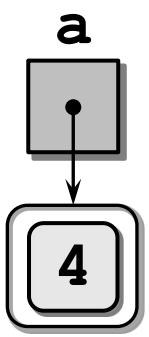


Linked Lists

Step 1: Store the Data

```
class Node:
    def __init__(self, data):
        self.data = data
```

When I do the command **`a = Node(4)`**
I create the following object...



`a` is *an alias*, or *another name for* **`Node(4)`**

`a.data` and **`Node(4).data`** both refer to ... **`4`**

`a` is a *shallow copy* of **`Node(4)`**

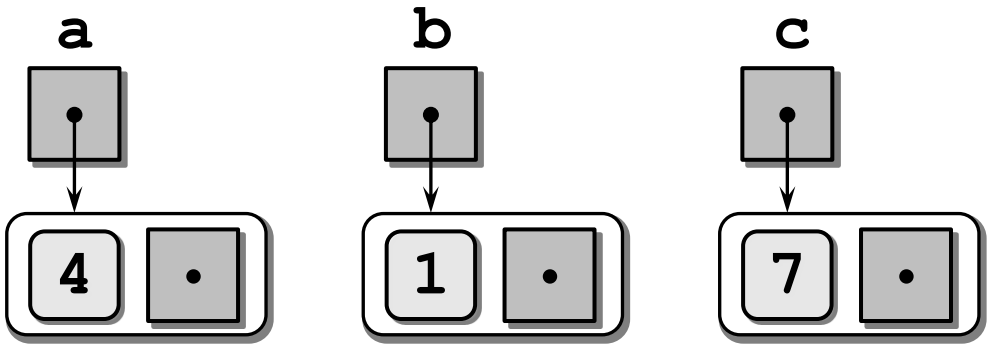
Linked Lists

Step 2: Add a Second Attribute *Next*

- add a second attribute to the Node class, called *next*
- set its value to None

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
a = Node(4)
b = Node(1)
c = Node(7)
```



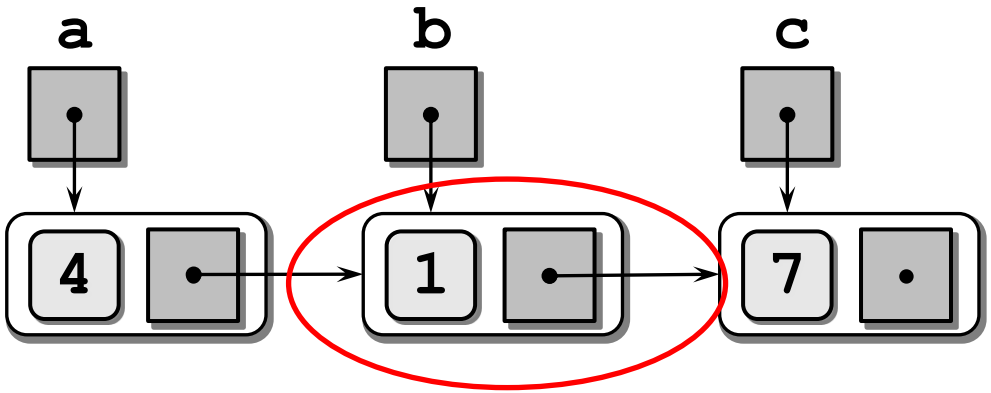
Linked Lists

Step 2: Add a Second Attribute *Next*

- *Rather than* have the *variables* **`a`**, **`b`**, and **`c`** to refer to the objects *have* the *objects* point to each other

```
a.next = b
b.next = c
```

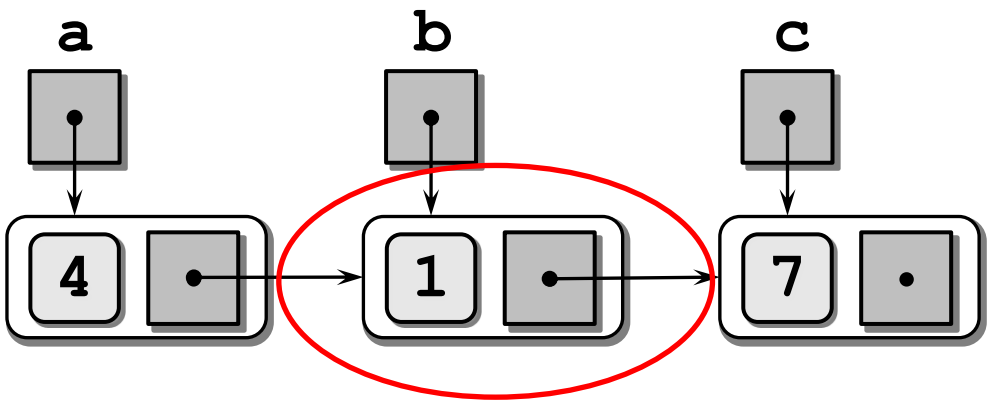
- **`a.next`** is now an *alias* or *refers to the same object* that **`b`** does



Linked Lists

Step 2: Add a Second Attribute *Next*

- *After I do the command "***`a.next = b`***" then both* **`a.next`** *and* **`b`** *refer to the same thing and anywhere that I use* **`b`** *I can also use* **`a.next`**

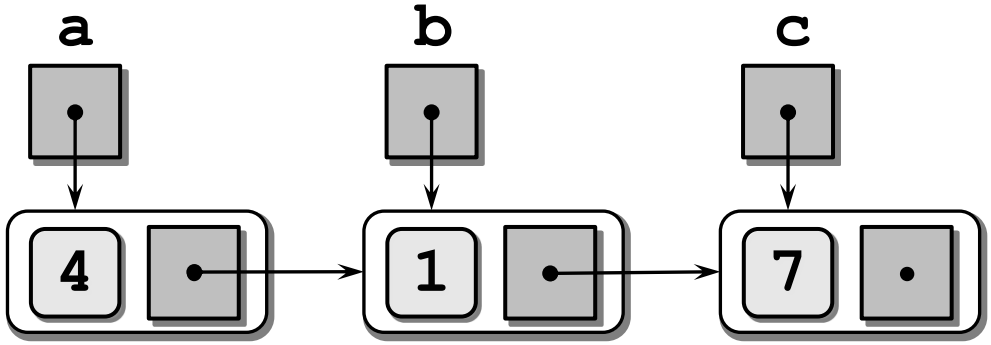


- So since **`b.data`** equals 1, then **`a.next.data`** also equals 1.

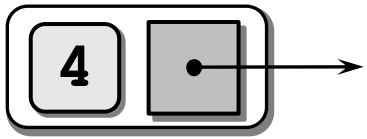
Linked Lists

Step 2: Add a Second Attribute *Next*

- In the diagram below if we say **`h = a`**

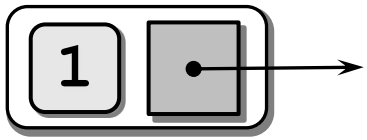


`h` is another name for



`h.data` is another name for

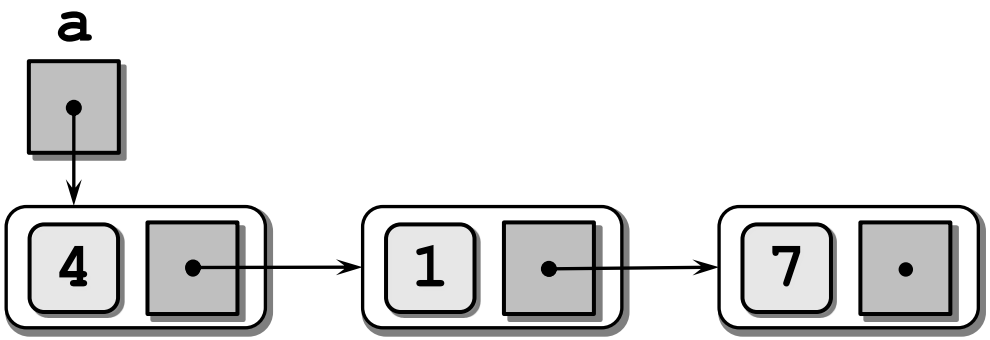
`h.next` is another name for



Linked Lists

Step 2: Add a Second Attribute *Next*

- We can now *get rid of* variables **`b`** and **`c`**, because we no longer need them
- Refer to the next object in the list using the *next* attribute, i.e. use **`a.next`** instead of **`b`**
- Now you *no longer have* one variable for each data item in the list.



Key Concept

Pointer Semantics / Shallow Copy

- A *key* but *subtle* concept of last few slides is the idea of one object *pointing to* the next item, *referencing* the next item or being *another name* for that item.
- When I say **`a.next = b`** then they *both refer to the same object*.
- Anywhere I use name **`b`** I can also you use the name **`a.next`**. *They are equivalent* .
- This idea of one object pointing to another *will come up again and again when discussing data structures*.

Linked Lists

Cautions when Implementing

- Always check that your code handles the following situations
  - an *empty list*
  - doing something (e.g. adding or removing) the *first element* in a list
  - doing something (e.g. adding or removing) the *last element* in a list

The Sequence ADT

Properties of a Sequence ADT

- Data values: a linear collection of items
- Constructor: `Sequence()`

Adding Items

- `insert()`, `append()`, `extend()`

Removing Items

- `pop()`, `remove()`

Basic Operations

- `str()`, `len()`, `reverse()`, `__getItem__()`
  also known as '='

The Sequence ADT

Linked List Implementation : Attributes

- **`_head`**, of type **`Node`**, referencing the *first value* in the sequence
- **`_len`**, of type **`int[ ≥0 ]`**, storing the *number of values* in the sequence
- **`_len`** is *optional* but makes finding the length O(1) rather than O(*n*)
- as values get added or removed from the list, update **`_len`**

The Sequence ADT

Basic Operations

- **`Sequence(self)`**
  construct an empty sequence: _len=0, _head=None
- **`str(self)`**
  Return a string representation of all the values in the sequence: e.g. "**`[4,1,7]`**"
- **`getItem(self, i)`**
  Get the *i*th item in a list: follow the *next* field *i* times and return the item at that location
- **`len(self)`**
  Return the length of (number of items in) the sequence.

The Sequence ADT

A Difficult Case: Remove an Item by Value

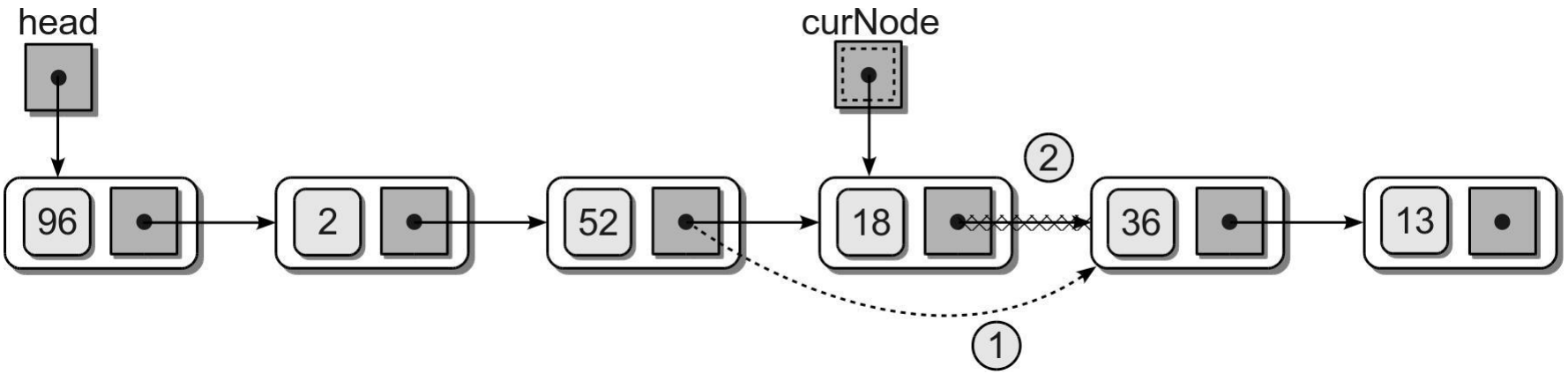
```
def remove(self, aValue):
    #advance through list until you find aValue
   curNode = self.head
   while (curNode is not None) and \
         (curNode.data != aValue):
      curNode = curNode.next
```

- E.g. to remove the value 18, advance through list until you find the value 18
- The order you evaluate the conditions in the while loop is important.

The Sequence ADT

Remove an Item by Value: The Problem

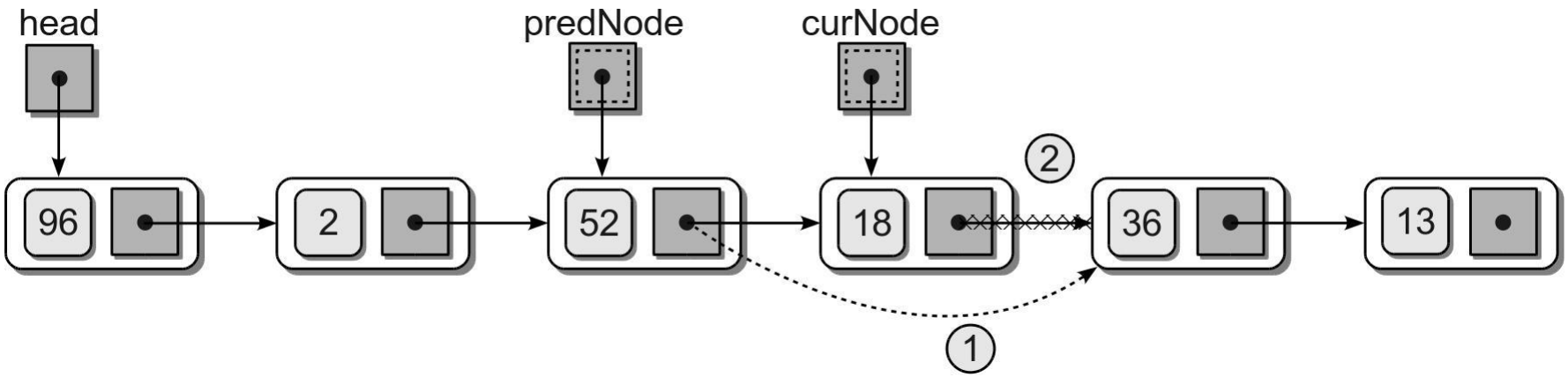
- The problem is once we find the node containing the value 18, we need to update the *next* pointer of the previous node (52) to point to 36
- i.e. we need to keep track of the previous node



The Sequence ADT

Remove an Item by Value

- We need a reference, *prevNode* to the *previous* Node



The Sequence ADT

Remove an Item by Value

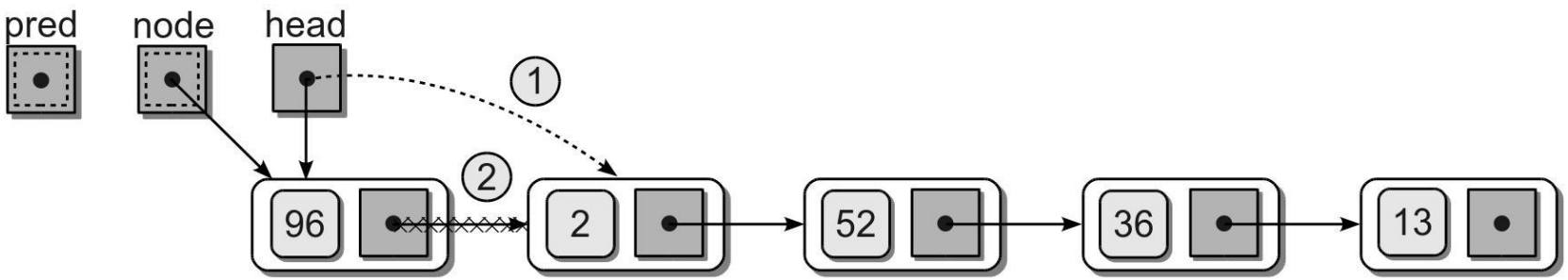
```
def remove(self, aValue):
    # advance through list until you find aValue
    curNode = self.head
    prev = None
    while (curNode is not None) and \
          (curNode.data != aValue):
        prev = curNode
        curNode = curNode.next
    if curNode != None:
        prev.next = curNode.next
```

Similar to slide 135 except where there are arrows.

The Sequence ADT

Special Case: Removing first value

- We need to update **`head`** in this case, not the **`next`** field of a **`Node`**




```
if head.data == aValue:
    head = head.next
```

The Sequence ADT

Removing Values By Position

`pop(self, index)`

- Find Node at position index-1
- Update its next field to its next.next
- Decrease the length by 1
- Special case: pop(0) (head changes, rather than a next field)

The Sequence ADT

Adding to a Sequence

`append(self, anItem)`

- Task: append anItem to the end of the list
- find the last **`Node`** in the list
- update its **`next`** field to **`Node(anItem)`**
- update length
- Special case: empty sequence

The Sequence ADT

Adding to a Sequence

```
insert(self, index, anItem)
```

- Task: insert anItem after position index
- "skip over" index-1 Nodes, to have a reference to the index-1th Node, as its value is the one being changed
- update length
- Special case: insert at front

The Sequence ADT

Reversing a Sequence

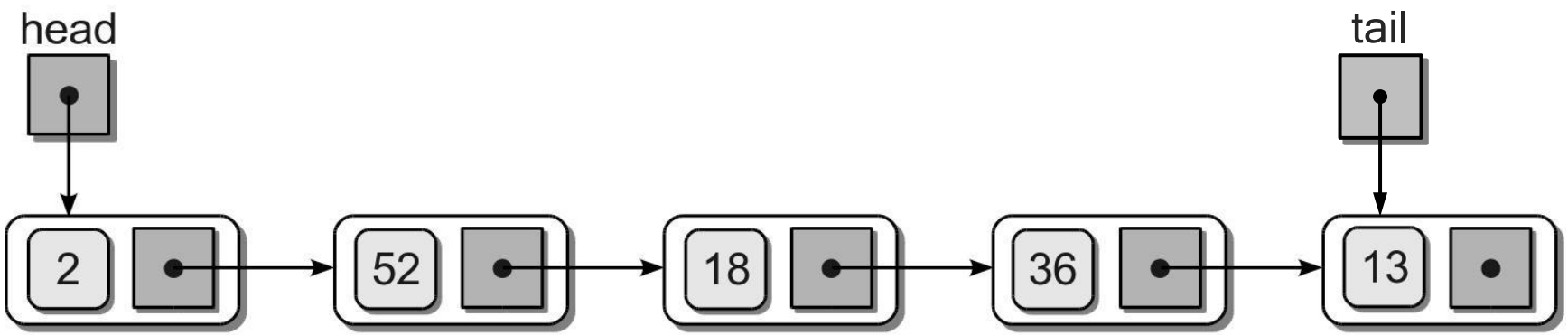
`reverse(self)`

- Task: reverse the order of items in a list
- Remember first item
- Pop first item
- Reverse "rest" of list (using recursion)
- Append old first item

Variations on Linked Lists

Tail References

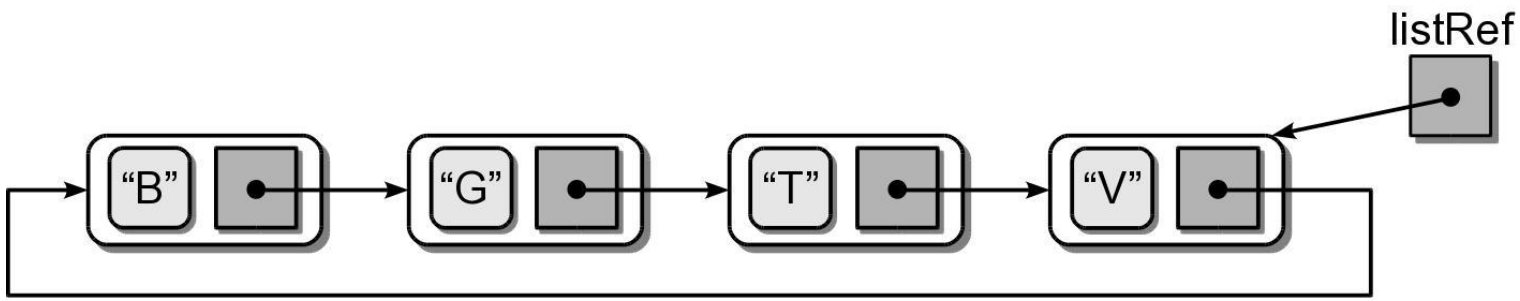
- Singly-linked list (SSL) with head and tail references
- What is the advantage?
  - Finding the last node for append/extend would then require O(1) operations instead of O(n).
  - Just need one more reference to store tail in order to get improved performance



Variations on Linked Lists

Circularly Linked Lists (CLL)

- Last item points to the first item



- Node is still useful for CLL- note that we use a reference to the tail rather than the head – why?

  - since tail.next is O(1) access to the head

Variations on Linked Lists

Circularly Linked Lists (CLL)

- a circular list of length 1 is a node that points to itself
- *adding to the front*: O(1) using listRef as tail reference

```
listRef.next = Node(aValue, listRef.next)
```

- *removing from the front*: O(1)

```
listRef.next = listRef.next.next
```

- *adding to the tail:* O(1)

```
listRef.next = Node(aValue, listRef.next)
listRef = listRef.next
```

Variations on Linked Lists

Circularly Linked Lists (CLL)

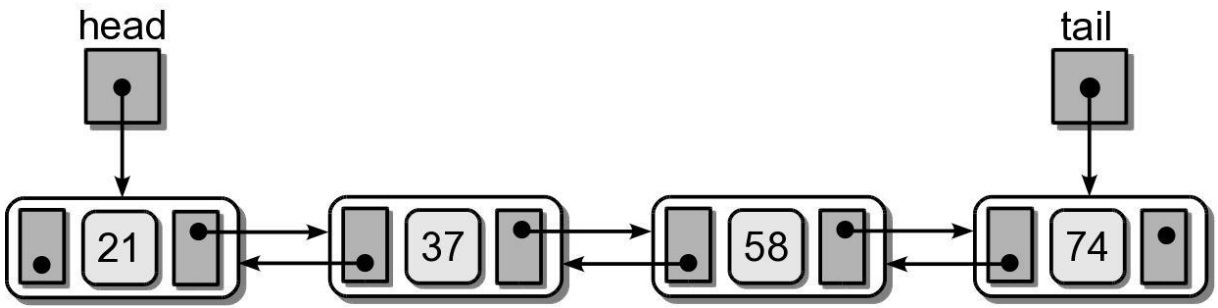
- Removing from the tail is still slow, O(n), since we need a reference to the second last item in the list
- How does a traversal/search work?

```
end = listRef # remember end of list
curNode = listRef.next
while curNode != end:
    # traverse or search...
    curNode = curNode.next
```

Variations on Linked Lists

Doubly-Linked Lists (DLL)

- Two references per node: **`prev`** and **`next`**
- Two reference per list: **`head`** and **`tail`**



- Can traverse the list in either direction

Variations on Linked Lists

Doubly-Linked Lists (DLL)

- Need a DoubleNode, with prev and next references,
- Can go from *head* -> *tail*, following *next*
- Can go from *tail* -> *head*, following *prev*
- Can visit each node, starting anywhere in the list
- Use *None* for *tail.next* or *head.prev*
- Can have variations on these as well, with circularly doubly linked lists.

Variations on Linked Lists

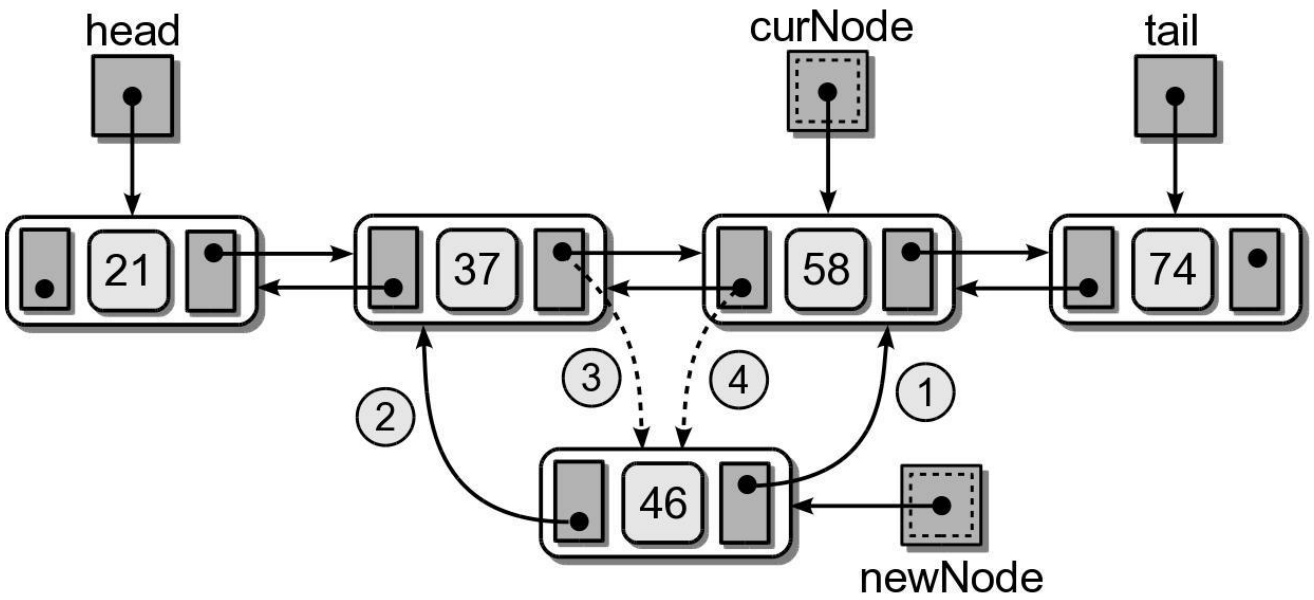
Doubly Linked: Insert

- Locate the position for the new node, then connect the links.
- Connections must be made in a specific order.
- There are three cases
  1. insert in the *middle*
  2. insert at the *beginning*
  3. insert at the *end*

Variations on Linked Lists

Doubly Linked: Insert

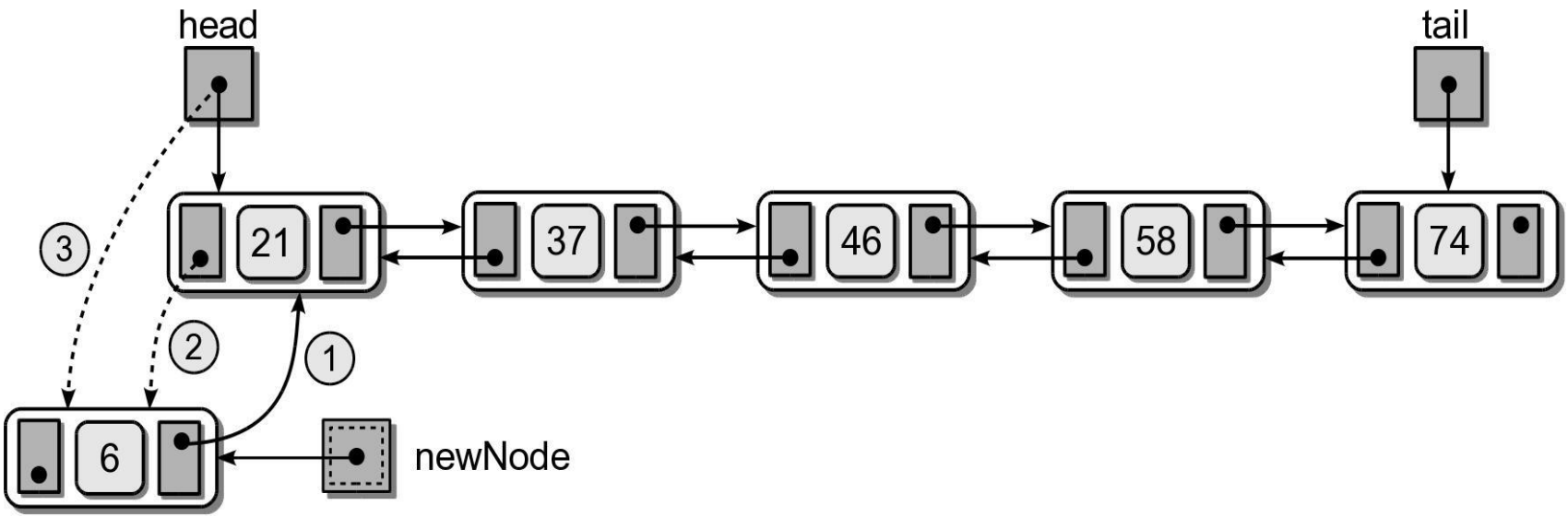
Case 1: Insert in the *middle*



Variations on Linked Lists

Doubly Linked: Insert

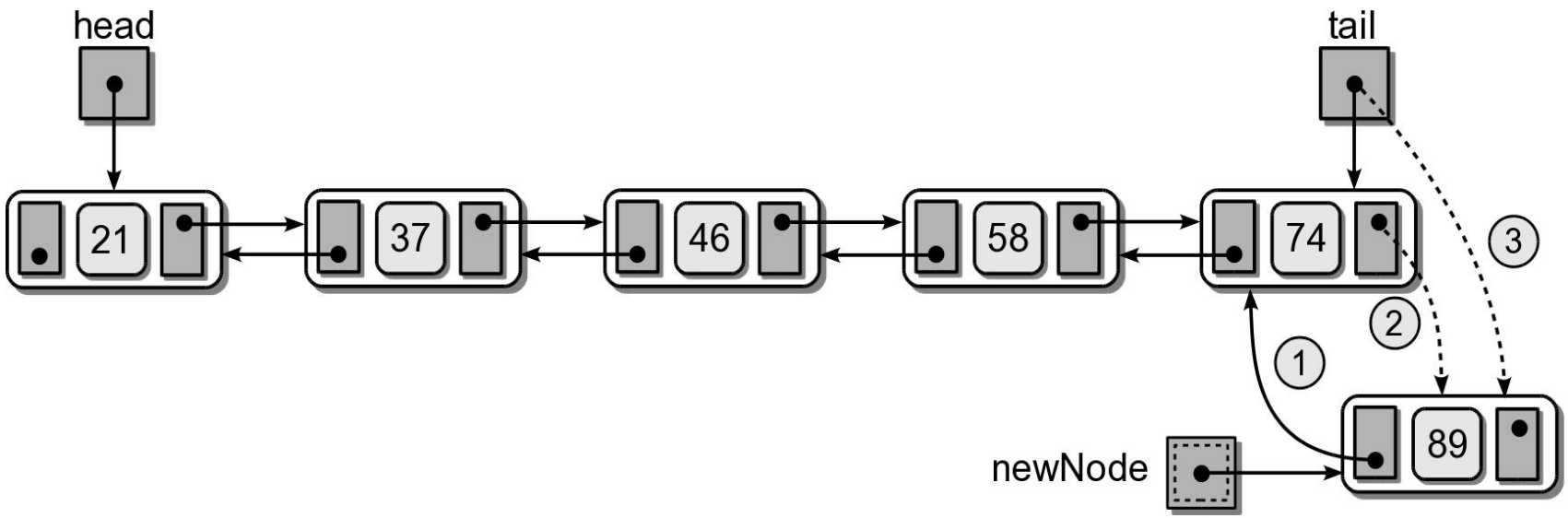
Case 2: Insert at the *beginning*



Variations on Linked Lists

Doubly Linked: Insert

Case 3: Insert at the *end*



Variations on Linked Lists

Memory Requirements

- Suppose integers, floats, strings and references require up to 4 bytes (old) or 8 bytes (new) of memory. Compare:

  - Array-based list with no additional capacity

  - Array-based list with half unused

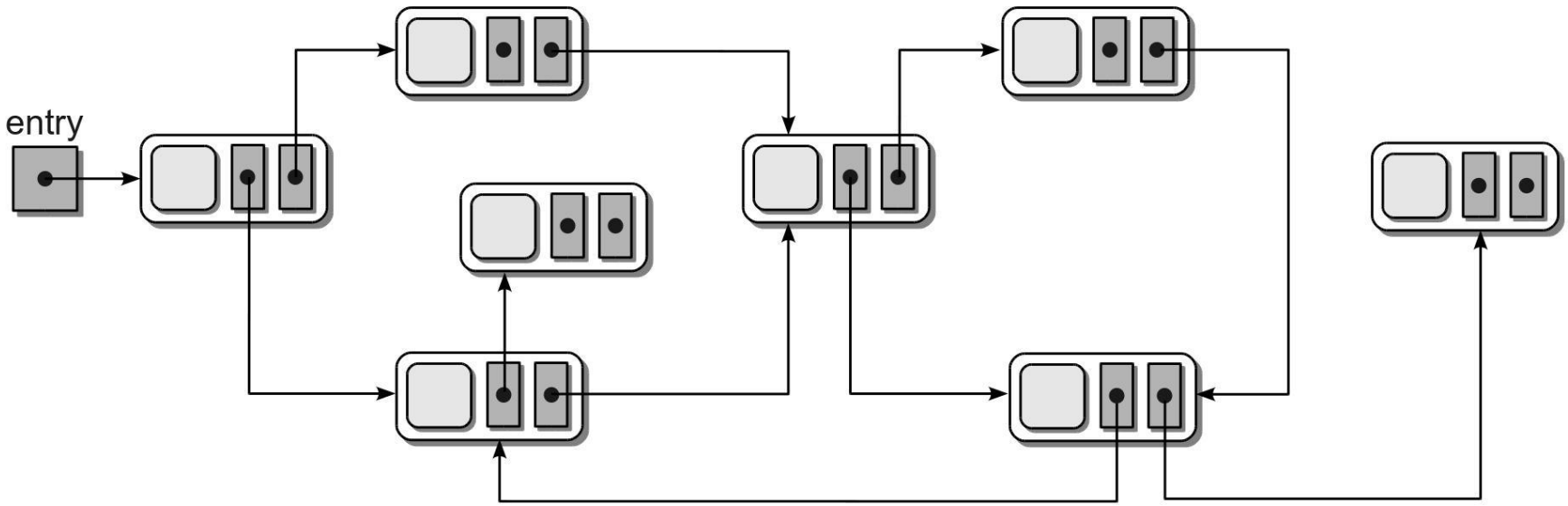
  - SLL-based

  - CLL-based

  - DLL-based

Variations on Linked Lists

- *They don't even need to be linear!*



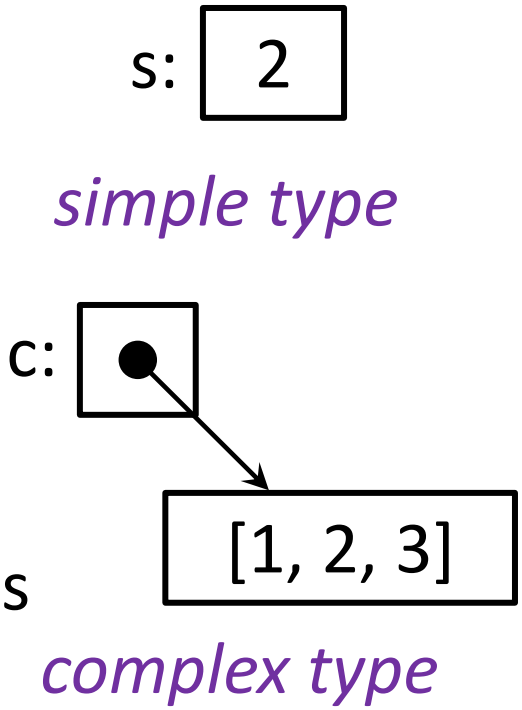
Assigning Simple vs. Complex Types

The Assignment Operation

When you assign one variable (or object) to another variable, such as

```
a = b
```

what happens depends on whether b represents a simple data type such as int or float or complex data type such as a Python class.



see: example11_reference.py

Assigning Simple vs. Complex Types

Assigning a Simple Type

a = 1

put the *value* 1 into memory location a

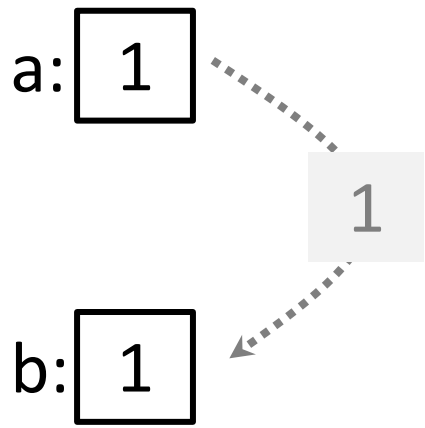
a: 1

Assigning Simple vs. Complex Types

Assigning a Simple Type

a = 1

put the *value* 1 into memory location a



b = a

copy the *value* in memory location a to memory location b

Assigning Simple vs. Complex Types

Assigning a Simple Type

a = 1

a: 3

put the *value* 1 into memory location *a*

b = a

b: 1

copy the *value* in memory location a to memory location b

a = 3

put the *value* 3 into memory location a

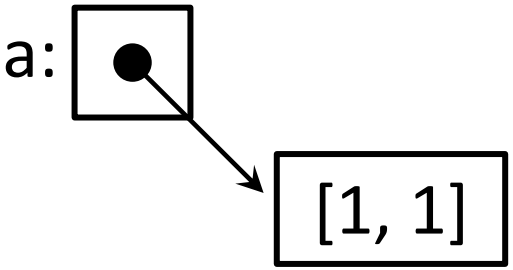
Assigning Simple vs. Complex Types

Assigning a Complex Type

a = [1, 1]

put the *location* of the list

into memory location a



Assigning Simple vs. Complex Types

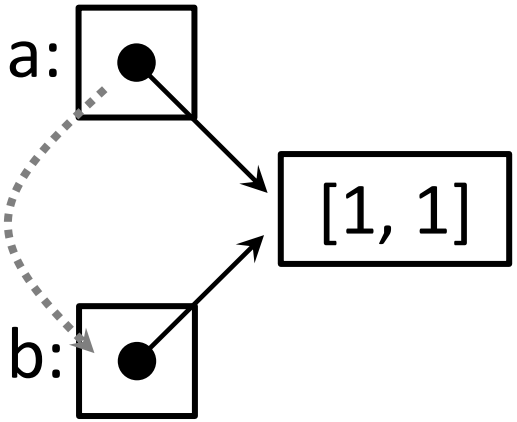
Assigning a Complex Type

a = [1, 1]

put the *location* of the list into memory location a

b = a

copy the *location* of the list into memory location b



Assigning Simple vs. Complex Types

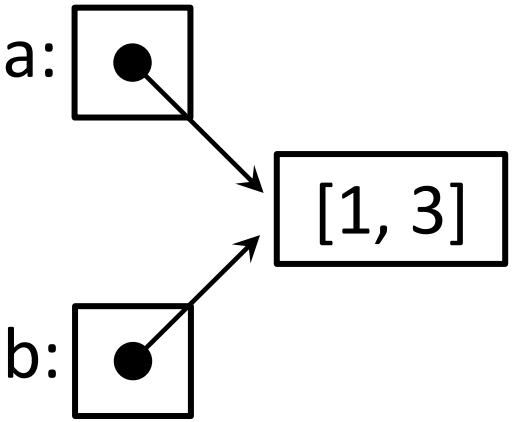
Assigning a Complex Type

a = [1, 1]

put the *location* of the list
into memory location *a*

b = a

copy the *location* of the list
into memory location b



a[1] = 3

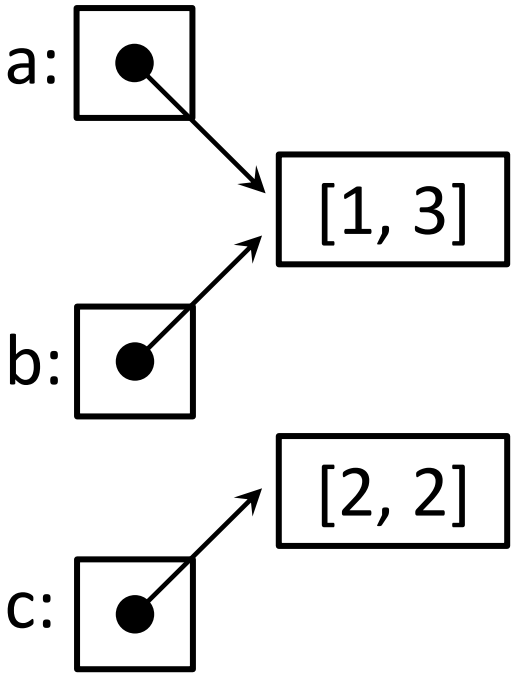
change the *value* of the list whose
location is stored in memory location a

Assigning Simple vs. Complex Types

Assigning a Complex Type

c= [2, 2]

put the *location* of a 2nd list into memory location c



Assigning Simple vs. Complex Types

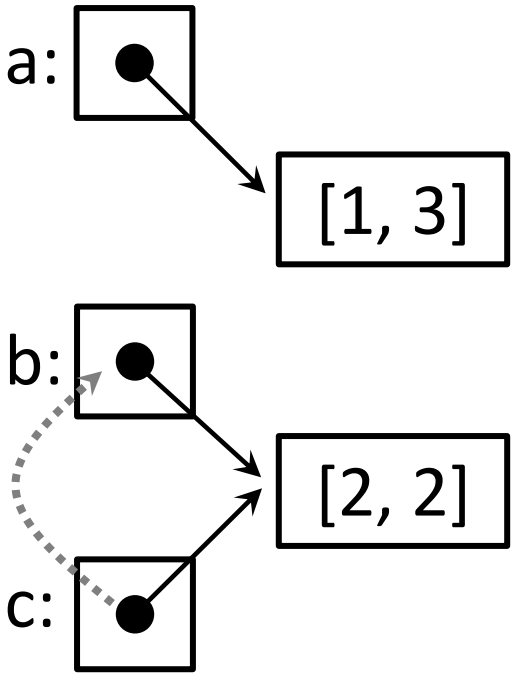
Assigning a Complex Type

c= [3, 3]

put the *location* of a 2nd list into memory location c

b = c

copy the *location* of the 2nd list into memory location b



Assigning Simple vs. Complex Types

Assigning a Complex Type

c= [3, 3]

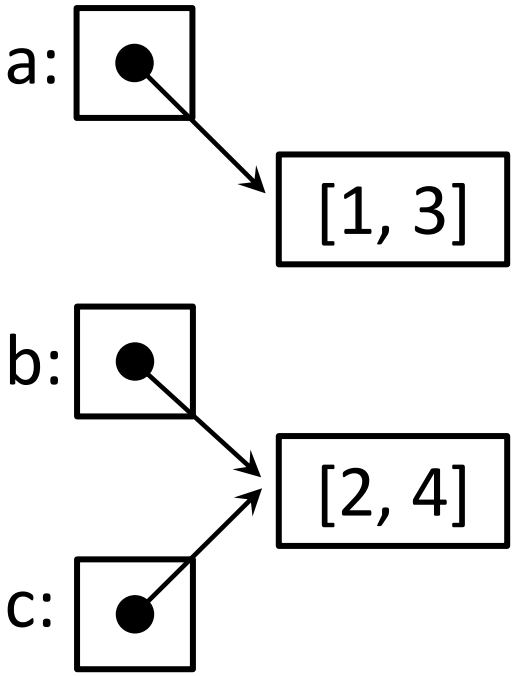
put the *location* of a 2nd list into memory location c

b = c

copy the *location* of the 2nd list into memory location b

c[1] = 4

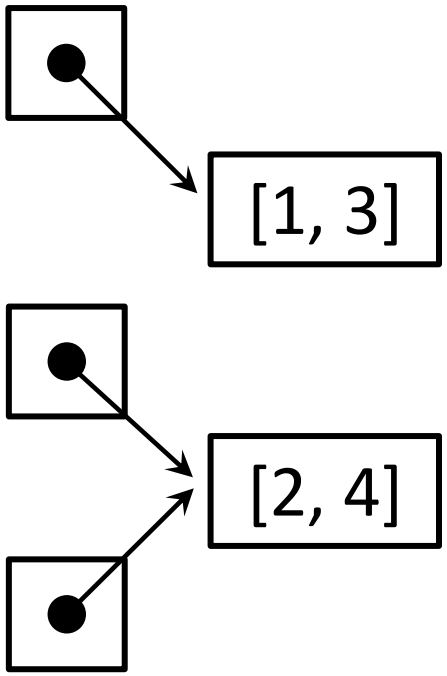
change the *value* of the list whose location is stored in memory location c



Assigning Simple vs. Complex Types

Referring to an Object

Much of what we will be doing with data structures will involve *referring to complex types (Python classes)* using variable names like head, tail, next, currentNode, prevNode, etc. to point to various parts of a data structure.



This method of dealing with objects is slightly different that what you are used to seeing with simple types like int and float.

Topic 7 – The Stack ADT

Key Concepts

- The Stack ADT
- Implementing the Stack ADT as a Python List
- Implementing the Stack ADT as a Singly Linked List

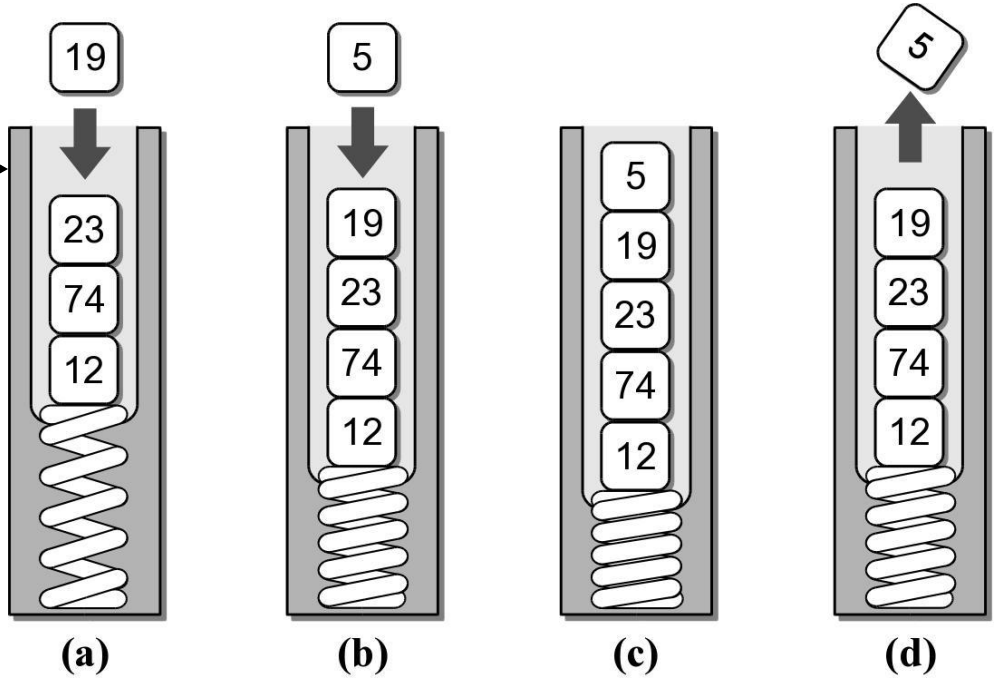
References

- Stacks, Sections 7.1 and 7.2, in the course text

The Stack ADT

- Linear collection of data values
- LIFO-protocol: *Last-In, First-Out*
- *Can only* add items to, or remove items from *the top*

Top of Stack →
- Add to top
- Remove from top



The Stack ADT

Operations

Key Restiction: Can only access the top item in a stack

- `Stack()` create an empty stack
- `isEmpty()` return True if the stack is empty
- `length()` return the number of items on stack
- `pop()` *remove* and *return top item* of nonempty stack
- `peek()` *return top item* of nonempty stack
- `push(anItem)` *add* item to top of stack

The Stack ADT

Example of Operations

Suppose **`aList = [5, 10, 15, 20]`**.

What is the value of **`aList`** after the following?

```
S = Stack()
while len(aList) != 0:
    S.push(aList.pop(0)) # remove 0th element
while not S.isEmpty():
    aList.append(S.pop())
```

Implementing the Stack ADT

Implementation #1: Python list

- Store stack contents using a *Python list*
- Which end of the list should be the top of the stack?
- How efficient are the operations?

  push is O(n) worst case (when the underlying array is full), O(1) amortized.

  pop is O(n) worst case (when underlying array less than 1/3 full, array is shrunk), O(1) amortized.

  other operations are O(1)
- see code at *example12_Stack_list_version.py*

Implementing the Stack ADT

Implementation #2: Linked List

- Store stack contents using a *singly linked list (SSL)*
- Should head or tail be the top of the stack?
- How efficient are the operations?

  push is O(1) worst case .

  pop is O(1) worst case .

  other operations are still O(1)
- see code at *example13_Stack_SLL_version.py*

Applications of ADT Stack

- Keeping track of functions calls and returns is a programming langauge
- Balancing brackets, html tags, etc.
- Backtracking algorithms (e.g. Solving a Maze example in Section 7.4 of course text)

Topic 8 – The Queue ADT

Key Concepts

- The Queue ADT
- Python List Implementation
- Circular Array Implementation
- Circular Link List Implementation

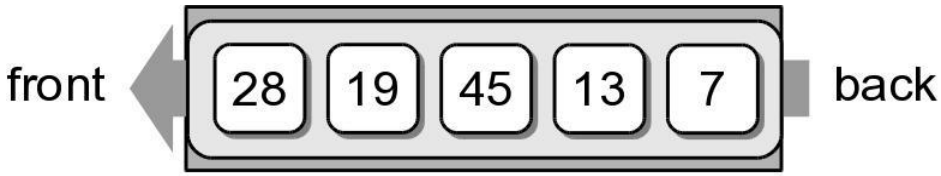
References

- Queues, Sections 8.1, 8.2, in the course text

The Queue ADT

- Linear collection of data values
- FIFO-protocol: First-In, First-Out
- Can only *add* items *to the back*
- Can only *remove* items *from the front*

Remove from the *front*



Add to the *back*

The Queue ADT

Operations

Key Restriction: The items are maintained in the order the are added.

- `Queue()` create an empty queue
- `isEmpty()` return if the queue empty or not
- `length()` return the number of items in the queue
- `dequeue()` remove and return front item from a nonempty queue
- `enqueue(anItem)` add anItem to the back of queue

The Queue ADT

Using the Queue ADT

Suppose **`aList = [5, 10, 15, 20]`**. What is the value of **`aList`** after the following executes?

```
Q = Queue()
for x in aList:
    Q.enqueue(x)
start = len(Q)-1
while not Q.isEmpty():
    aList[start] = Q.dequeue()
    start -= 1
```

Python List Implementation

Implementation #1: Python list

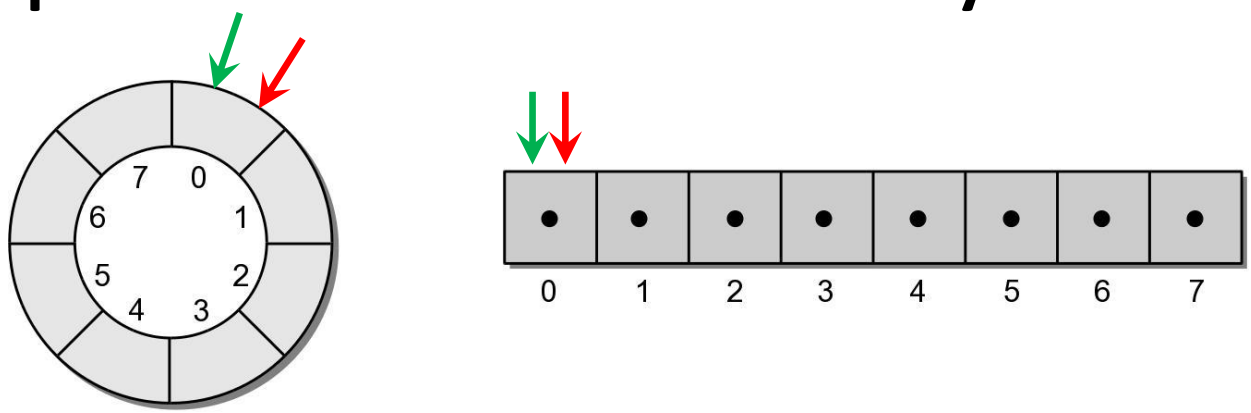
- Store Queue contents using a *Python list*
- Which end of the list should the front/back of the queue be?
- How efficient are the operations?

  enqueue() – O(n) worst case (i.e. enlarging underlying array when adding to the end

  dequeue() – O(n) worst case (shuffling when removing from beginning of list)
- see *example14_Queue_list_version.py*

Circular Array Implementation

Implementation #2: Circular array



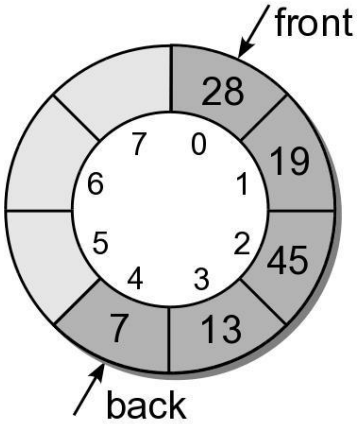
- Removes "shuffling" of array elements
- To enqueue or dequeue means moving a pointer
- But, array can reach "maximum capacity"

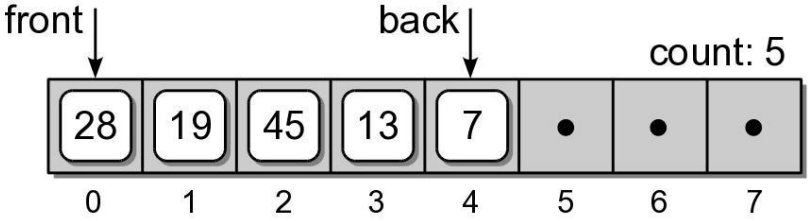
Circular Array Implementation

Organizing the Circular Array

How should the data be organized within the array?

- **count field** – number of items in the queue.
- **front and back pointers** – indicate the array elements containing the queue items.



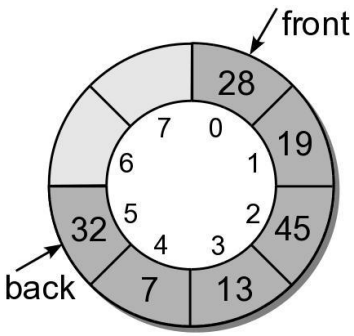


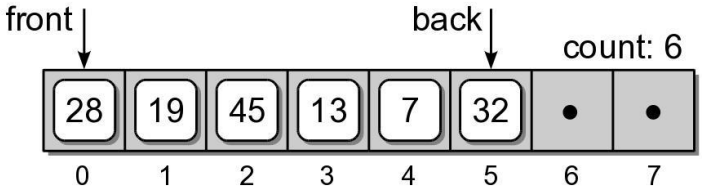
Circular Array Implementation

Enqueue for a Circular Array

enqueue(aItem):

- insert *anItem* at the position following *back*
- *back* is advanced by one position
- *count* is incremented by one.
- e.g. enqueue(32)



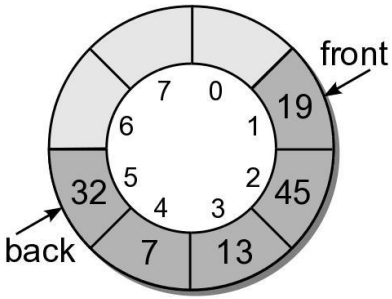


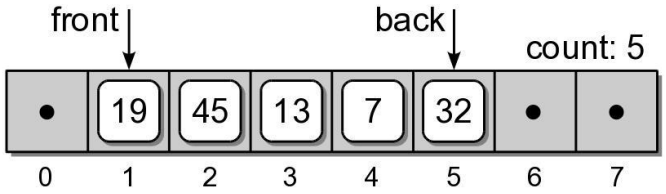
Circular Array Implementation

Dequeue for a Circular Array

dequeue():

- the value in the *front* position is saved
- *front* is advanced by one position.
- *count* is decremented by one.
- e.g. dequeue()

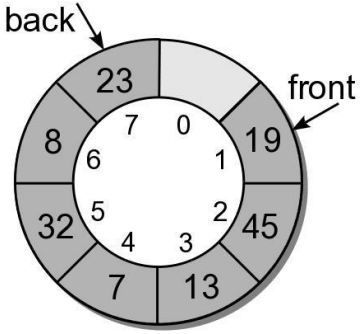


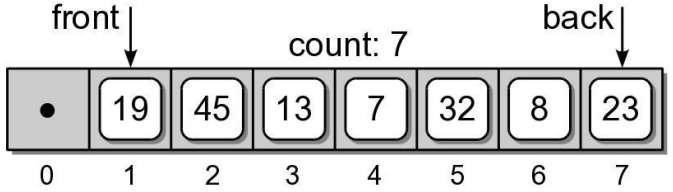


Circular Array Implementation

Enqueue and Wrap Around

- Suppose we enqueue 3 more items 8, 23, and 39?
- First the array fills up to the end.

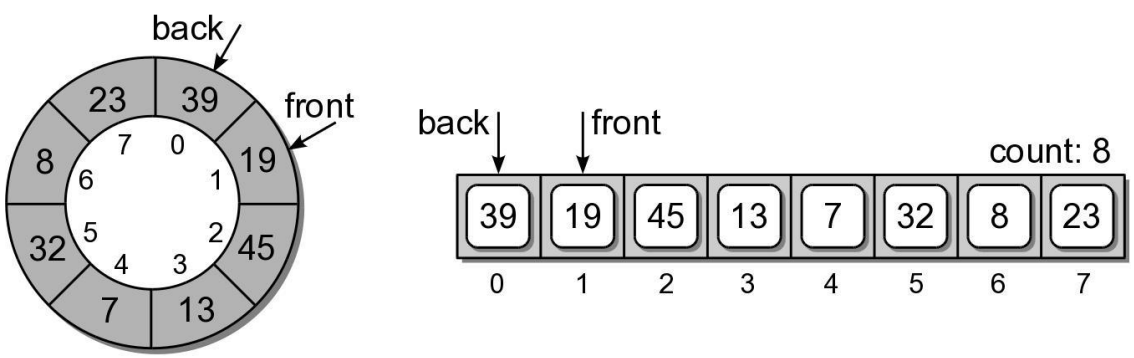




Circular Array Implementation

Enqueue and Wrap Around

- Then the queue "*wraps around*" and starts filling up from the beginning.
- Since we are using a circular array, the same steps are followed.
- But since *back* is at the end of the array, it wraps around to the *beginning* of the array.



Circular Array Implementation

Efficiency of operations

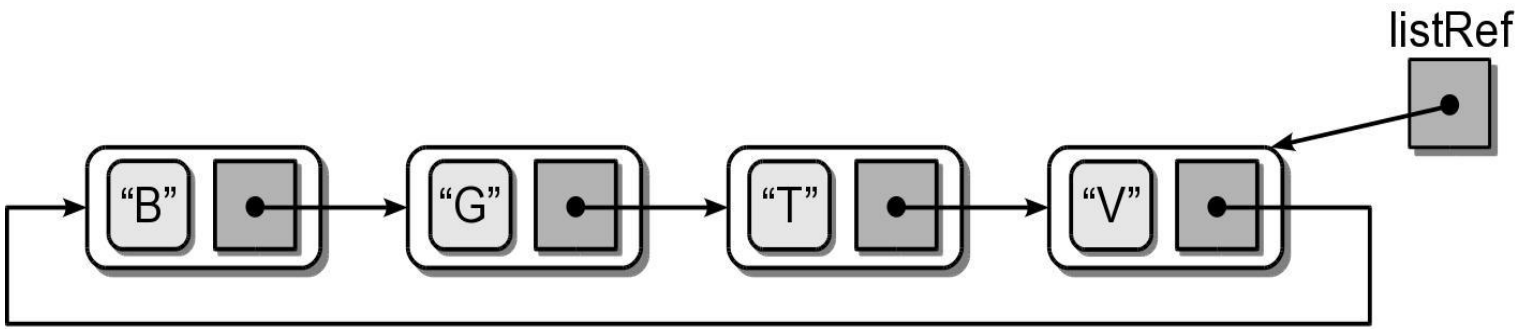
- *No shuffling* items to the left when you dequeue.
- *No copying* of elements when you enqueue and fill up the array.
- All operations are *fast*.

But …

- We now have *maximum queue capacity*

Circular Link List Implementation

Circular linked list implementation



- Do we point to the front or back of the Queue?

  tail = back of queue good for *dequeue*

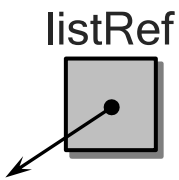
  head (tail.next) = front of queue good for *enqueue*
- Keep count of the number of items in the queue
- see *example15_Queue_CLL_version.py*

Circular Link List Implementation

Enqueue: 2 Cases

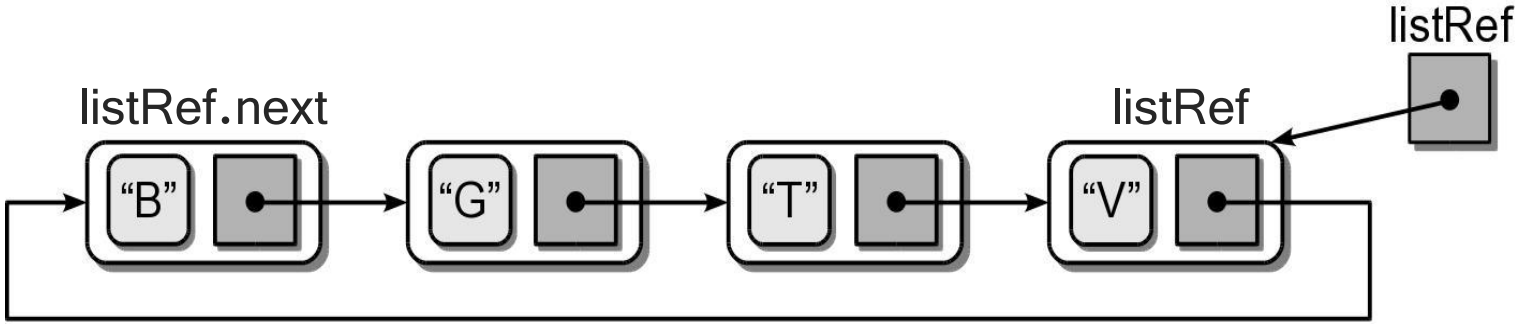
Special Case: Queue is empty

- new node points to itself



General Case:

- insert between listRef and listRef.next

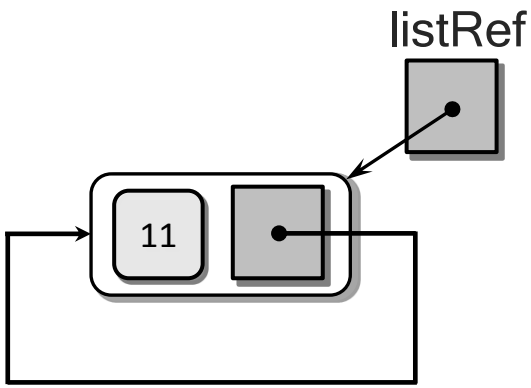


Circular Link List Implementation

Dequeue: 2 Cases

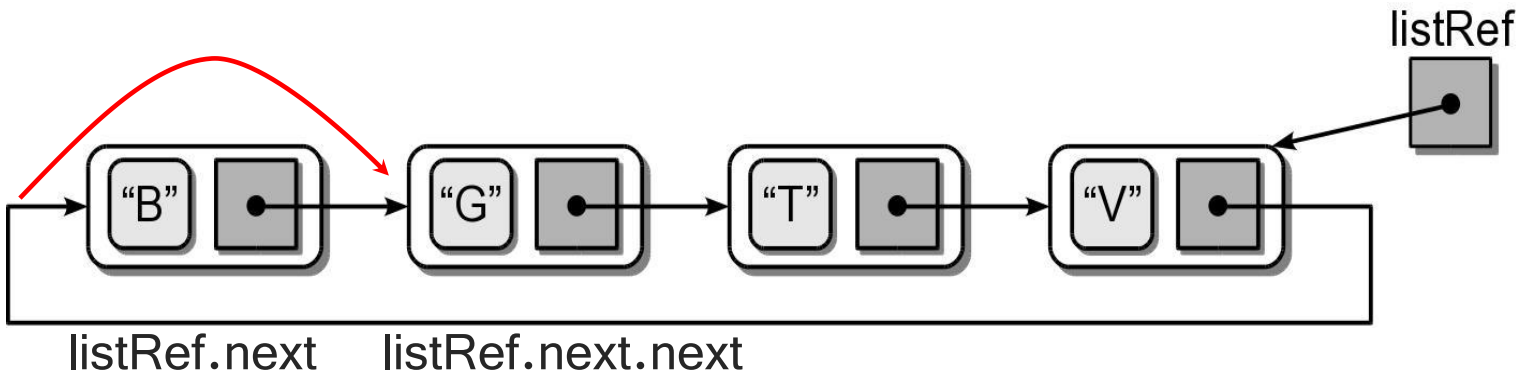
Special Case: Queue has one node

- listRef = None



General Case:

- listRef.next (head) points to listRef.next.next



Applications of ADT Queue

- Simulating real-world situations

  List of customers in a store

  List of banking transactions

- Consumer/Producer Relationships

- Breadth-first traversal of binary trees

  More on that later!

Topic 9 – The Priority Queue ADT

Key Concepts

- The Priority Queue ADT
- Types of Priority Queues
- Unbounded Priority Queue Implementation
- Bounded Priority Queue Implementation

References

- Priority Queues, Section 8.3 in the course text

Priority Queue ADT

What is a Priority Queue

- Linear collection of *items + priorities*
- Each item has an associated integer priority ≥ 0
- Two types:

  Bounded priority queue:

  - Each priority p satisfies $p \leq M$, for some upper bound $M > 0$, i.e. p is in $\{0, 1, 2, \ldots M\}$

  Unbounded priority queue

  - No upper bound on p

ADT Operations

Operations

Key: enqueue can specify a priority and the constructor a number of levels

- `PriorityQueue()` creates empty queue
- `PriorityQueue(numLevels)` create empty queue with priorities < `numLevels`
- `isEmpty()` same as Queue
- `length()` same as Queue
- `enqueue(item, priority)` adds item (with priority) to queue
- `dequeue()` removes item with maximum priority from nonempty queue

Using a Priority Queue

Using the Queue ADT

Q: Suppose **aList = [12,4,1,9]**. What is the value of **aList** after the following executes?

A: `aList = [12,9,4,1]`

```
PQ = PriorityQueue()
for x in aList:
    PQ.enqueue(x,x)
pos = 0
while not PQ.isEmpty():
    aList[pos] = PQ.dequeue()
    pos += 1
```

Implementing an Unbounded Priority Queue

Store items in FIFO order

- Enqueue – add to end
- Dequeue - determine maximum priority, and remove first value with that priority

Store items in increasing order

- Enqueue – add into sorted position
- Dequeue – remove from end

Implementing an Unbounded Priority Queue

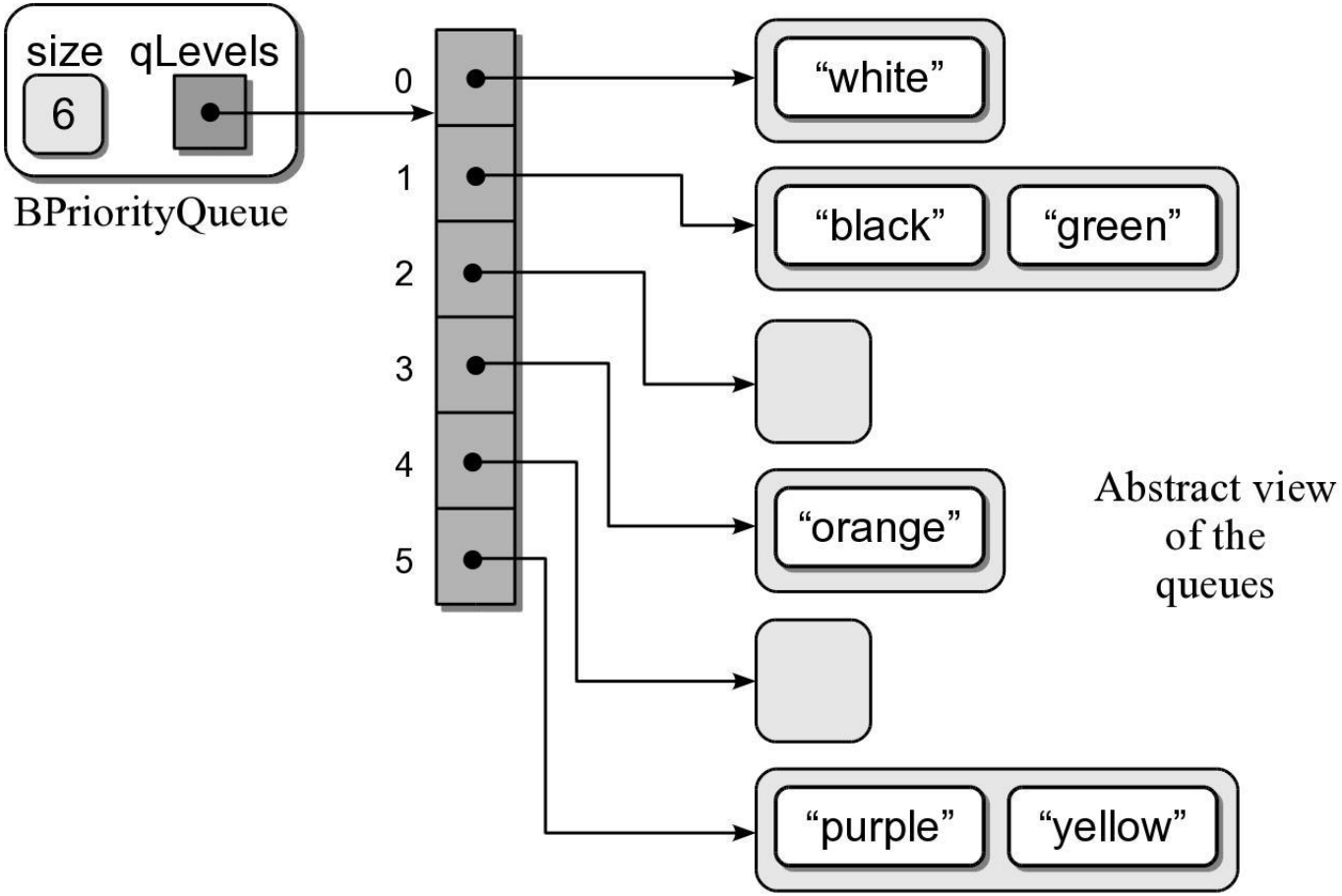
Python list

- FIFO order
  - Enqueue: O(n) worst
  - Dequeue: O(n)
- Sorted order
  - Enqueue: O(n)
  - Dequeue: O(n) worst
- see *example16_PQueue.py*

Linked list

- FIFO order
  - Enqueue: O(1)
  - Dequeue: O(n)
- Sorted order
  - Enqueue: O(n)
  - Dequeue: O(1)

Implementing a Bounded Priority Queue



Implementing a Bounded Priority Queue

Implementation Details

- Maintain a list of queues
- Each queue is indexed by priority *p*
- enqueue (item, priority)

  Enqueue to queue indexed by priority
- dequeue()

  Find maximum priority with non-empty queue

  Dequeue from that queue
- see *example17_BQueue.py*

Implementing a Bounded Priority Queue

Efficiency

- enqueue (item, priority)
  - Find appropriate queue is O(1)
  - Append to queue depends in implementation
  - it is O(1) for CLL
- dequeue()
  - Find maximum priority with nonempty queue O(*p*) but O(1) if *p* is fixed
  - Dequeue from that queue, O(1) for CLL

To study another implementation

- We must introduce/review Binary Trees

Topic 10 – Binary Trees

Key Concepts

- Tree Structure
- The Binary Tree
- Traversing a Tree
- Heaps

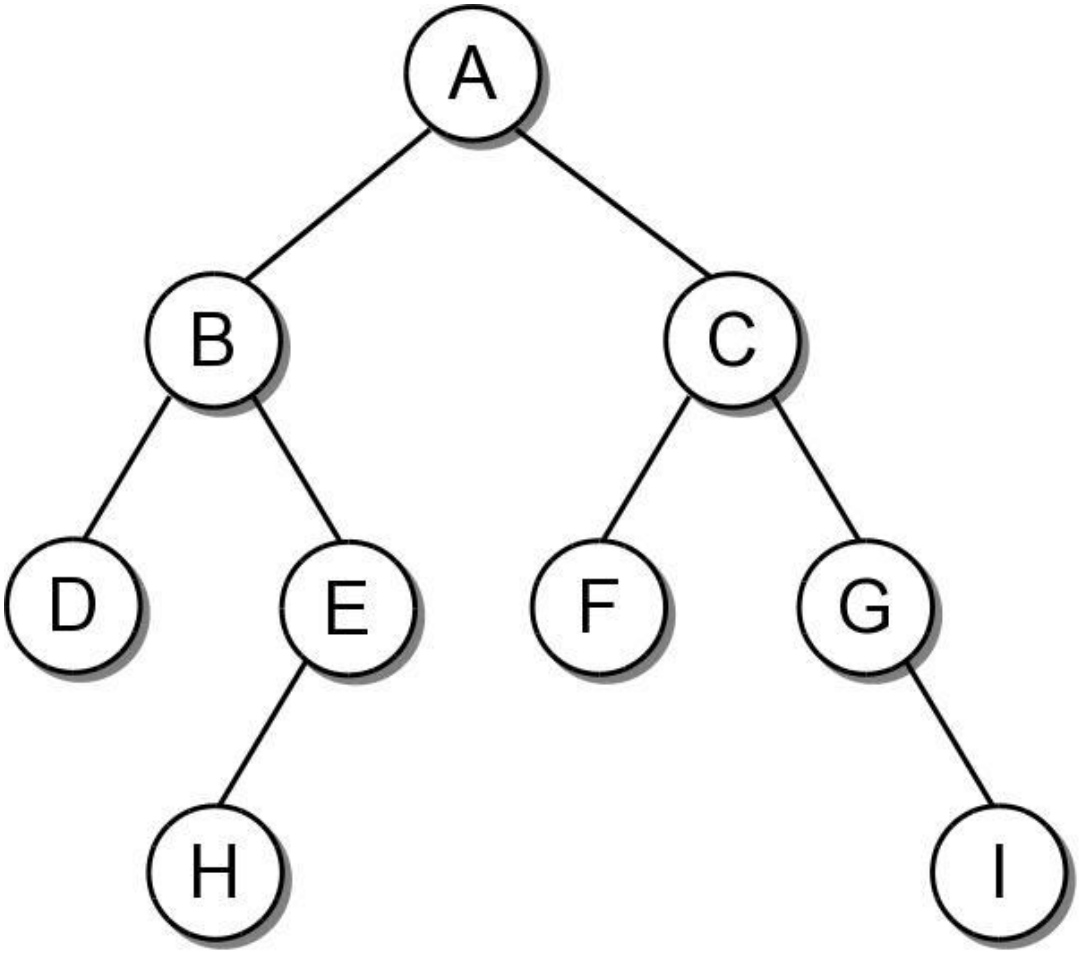
References

- Binary Trees, Section 13.1, 13.2, 13.4 in the course text

Tree Structure

Terminology

Root
Path
Parent
Child
Siblings
Interior Node
Leaf
Subtree
Descendants
Ancestors
Left Child
Right Child



Tree Structure

Other Important Terms

Size: number of nodes in the tree

Depth (level): Length of path from root to node

- 0 for the root
- 1 + depth of parent

Height: Number of levels in the tree

- 0 for the empty tree
- 1 + max(height(left), height(right))

Tree Structure

Properties of Binary Trees

- Given a binary tree is of size *n* and height h, then

  $h \leq n \leq 2^h - 1$

- For a *perfect binary tree* (all inner nodes have two children and all the leaves are at the same level)

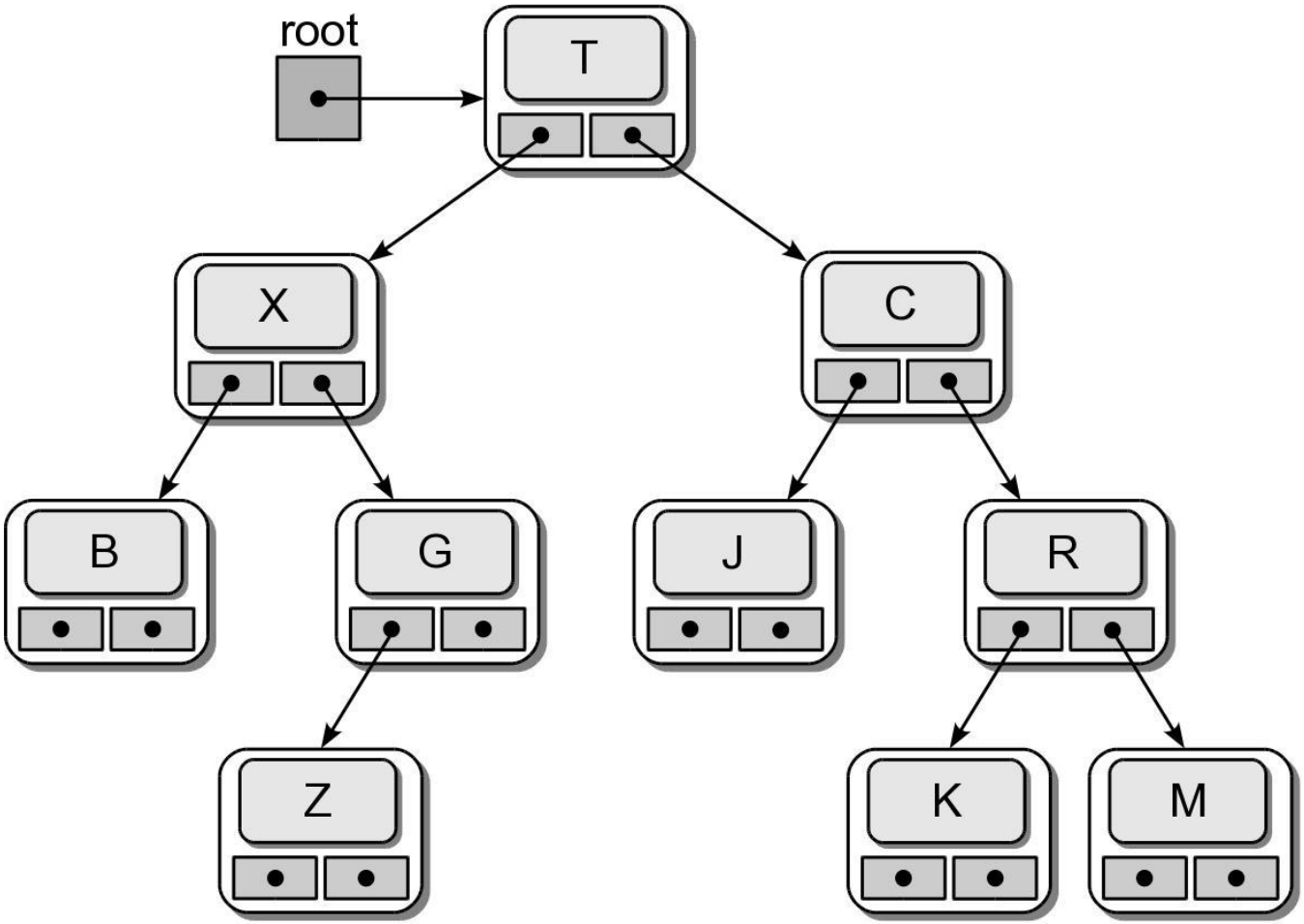
  $n = 2^h - 1$ nodes

  $h = \log_2(n + 1)$

- For a complete binary tree (perfect up to level *h*-1 and filled left to right a level *h*)

  $2^{h-1} \leq n \leq 2^h - 1$

The Binary Tree



The Binary Tree

Node Structure

- Implementation needs helper class Node or TreeNode, with left and right, as well as item value

```
class TreeNode:
    def __init__(self, item, left=None,\
                 right=None):
        self.item = item
        self.left = left
        self.right = right
```

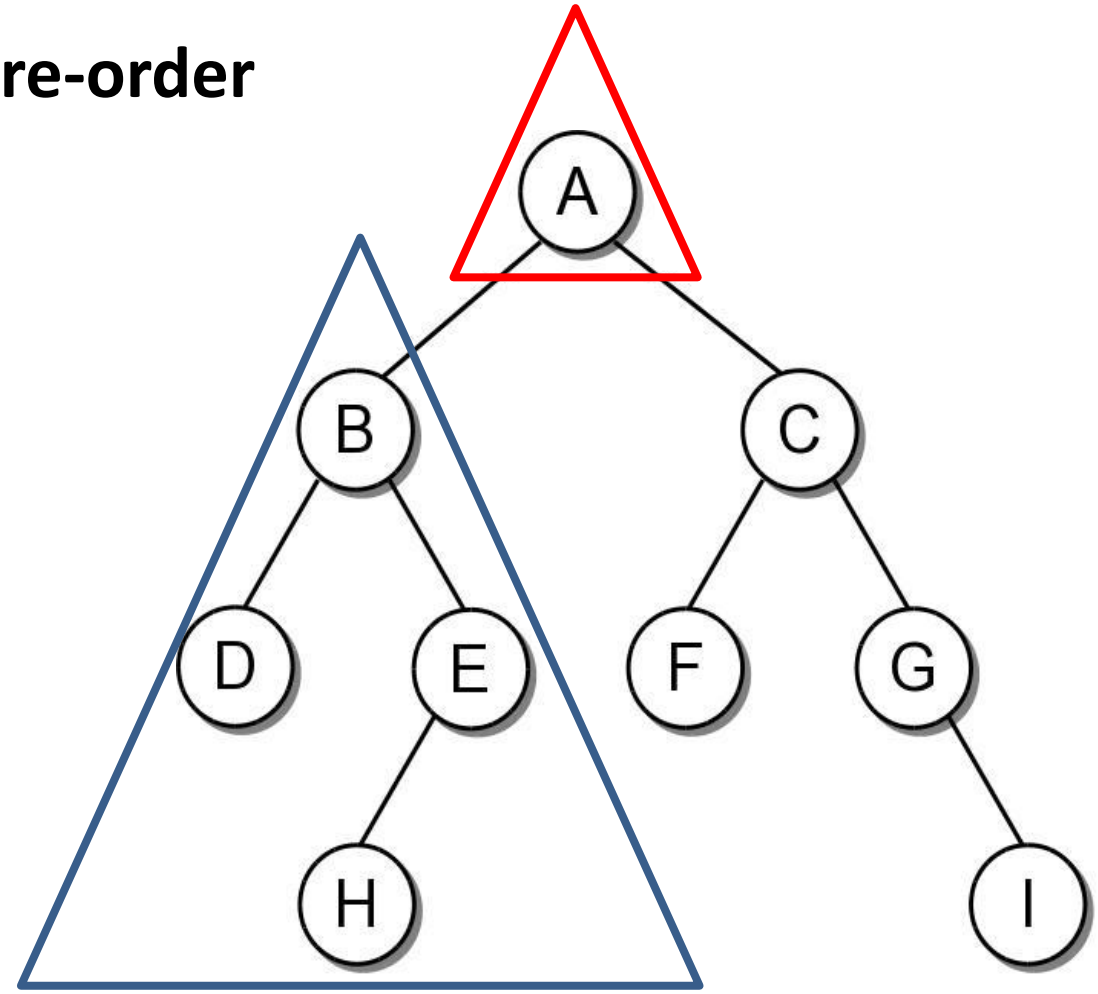
Traversals of Binary Trees

Depth First Search: Pre-order

```
def preorder(t):
  if t != None:
    print t.item
    preorder (t.left)
    preorder(t.right)
```

Pre-order:

A B D E H C F G I



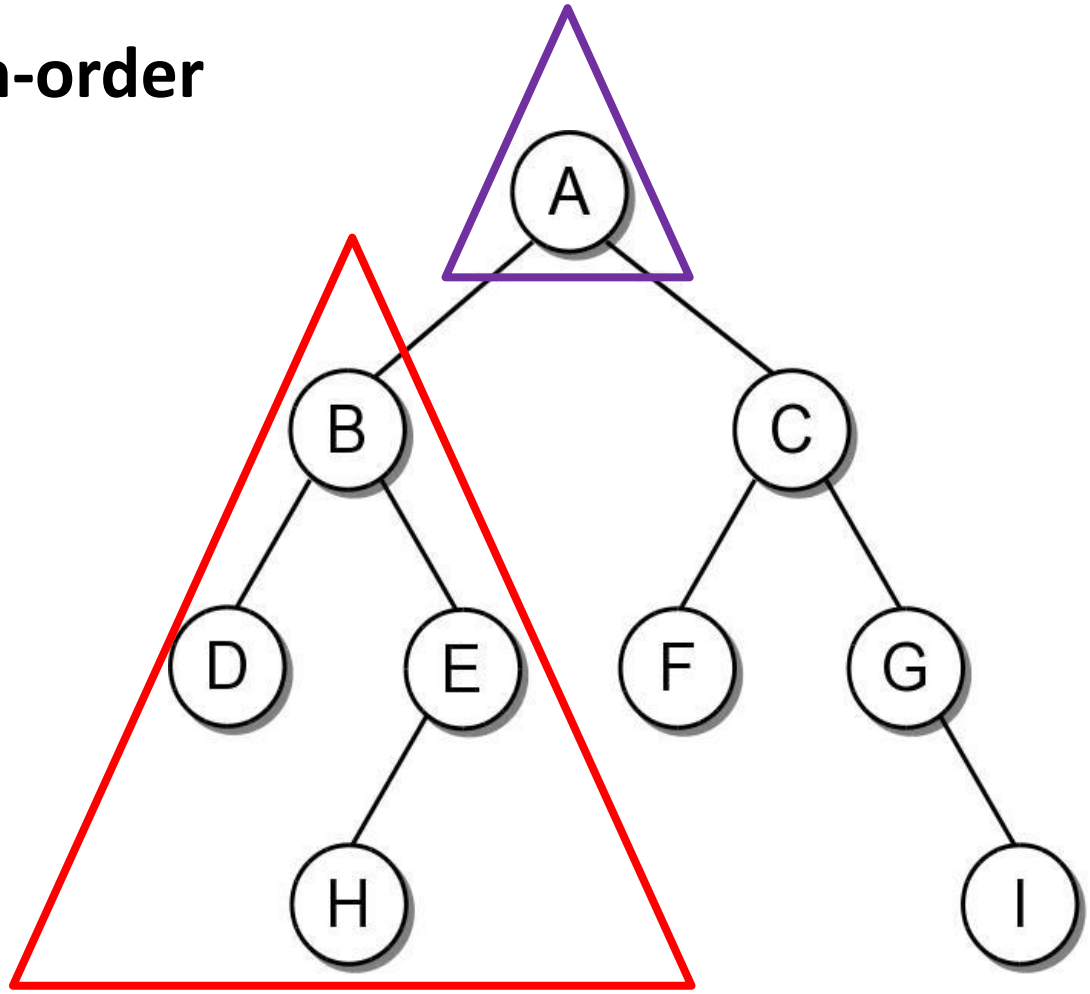
Traversals of Binary Trees

Depth First Search: In-order

```
def inorder(t):
  if t != None:
    inorder (t.left)
    print t.item
    inorder(t.right)
```

In-order:

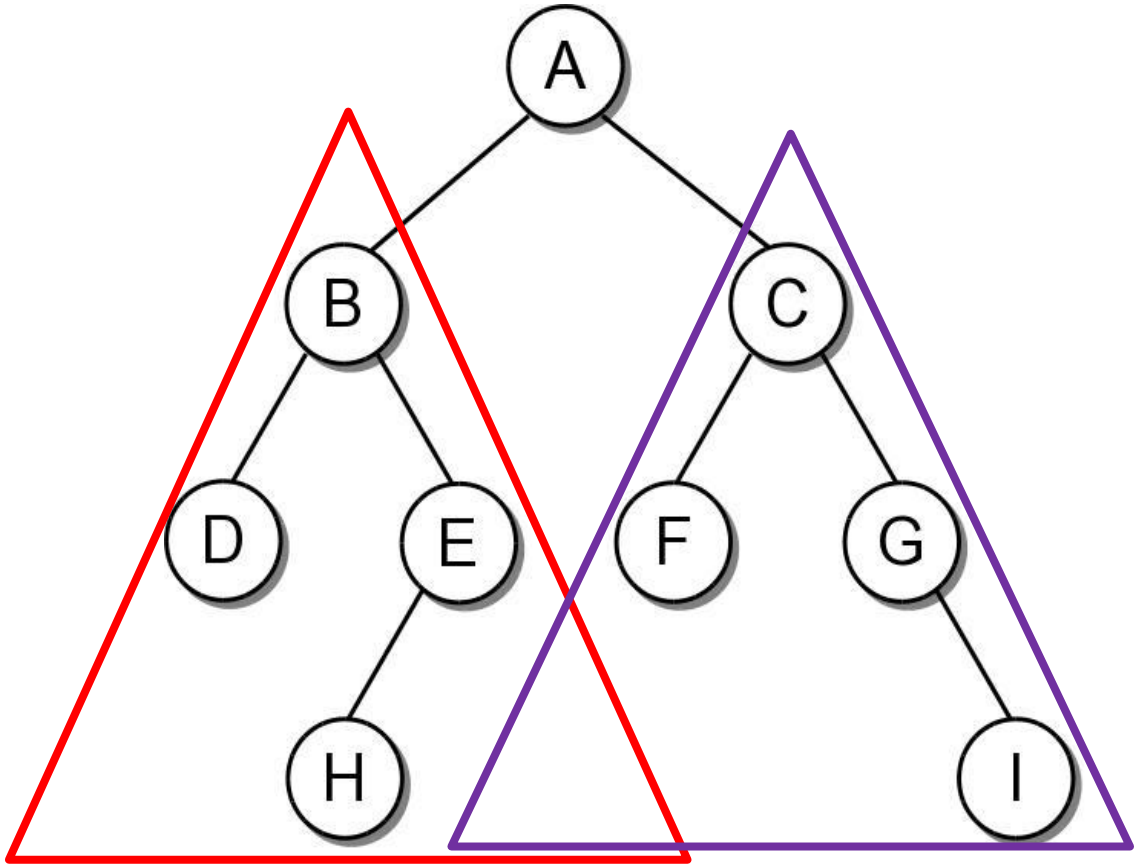
D B H E A F C G I



Traversals of Binary Trees

Depth First Search: Post-order

```
def postorder(t):
  if t != None:
    postorder (t.left)
    postorder(t.right)
    print t.item
```



Post-order:

D H E B F I G C A

Traversals of Binary Trees

Depth-first Traversals

For a non-empty tree:

- Visit the root value
- Recurse on left and right subtrees
- Different traversals are different orderings of these steps

  Pre-order – visit root before (pre) left, then right

  In-order – visit root in-between left and right

  Post-order – visit root after (post) left and right

Traversals of Binary Trees

Breadth-first Traversal

- Non-recursive
- When we visit a node
  add its children to a queue (FIFO)
  visit siblings before children

Traversals of Binary Trees

```
def breadthFirstTrav( bintree ):
  q = Queue()
  q.enqueue( bintree )

  while not q.isEmpty() :
    # Remove the next node from the queue
    node = q.dequeue()
    print( node.item )

    # Add the two children to the queue
    if node.left is not None :
      q.enqueue( node.left )
    if node.right is not None :
      q.enqueue( node.right )
```

Heaps

Operations

- Heap() construct an empty Heap
- insert() insert a new value

For Min-heaps

- removeMin() remove the minimum value

For Max-heaps

- removeMax() remove the maximum value

Heaps

Properties

Heap Shape Property

- heaps are complete binary Tree

Heap Order Property

- Max-heap: value of parent is *larger than* that of its children
- Min-heap: Value parent is *smaller than* that of its children

Heaps

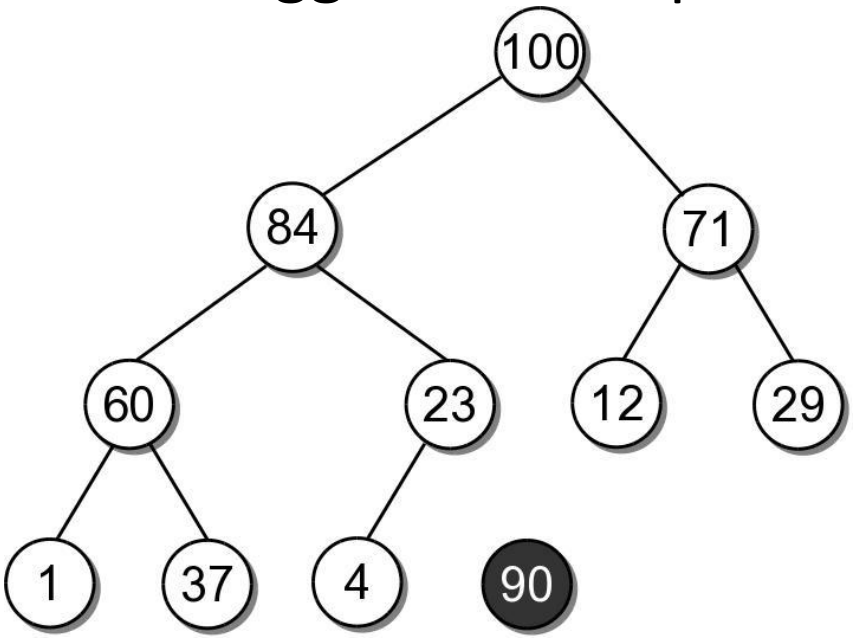
Insert

e.g. insert (90)

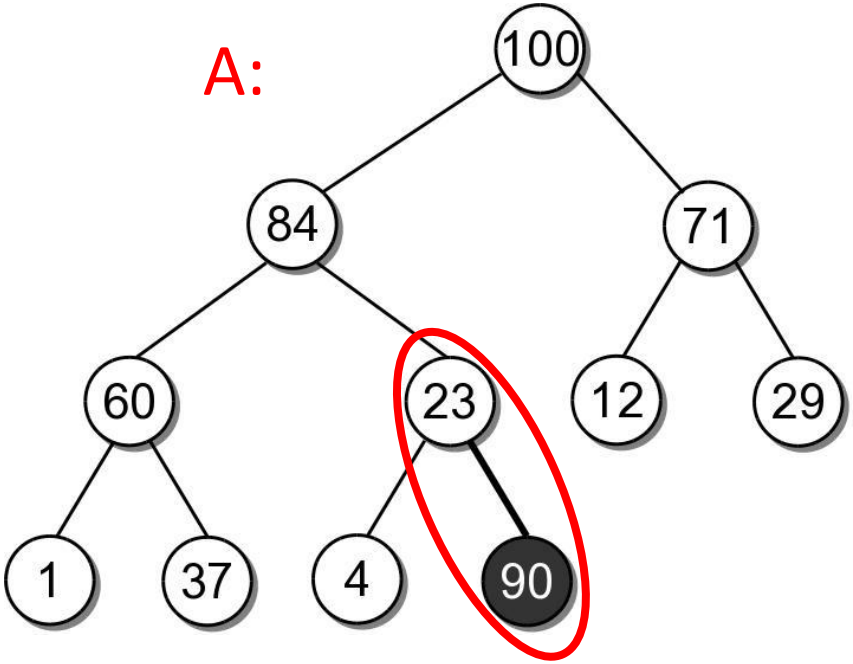
A: Place new item, 90, in the leftmost available leaf.

Swap with parent to preserve max (or min) relationship.

90 is bigger than its parent 23



A:



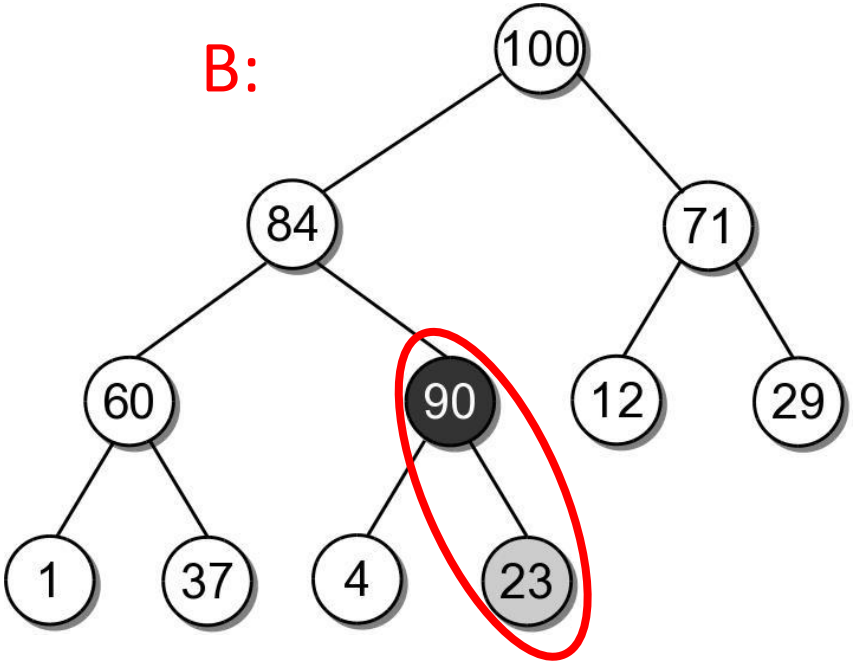
Heaps

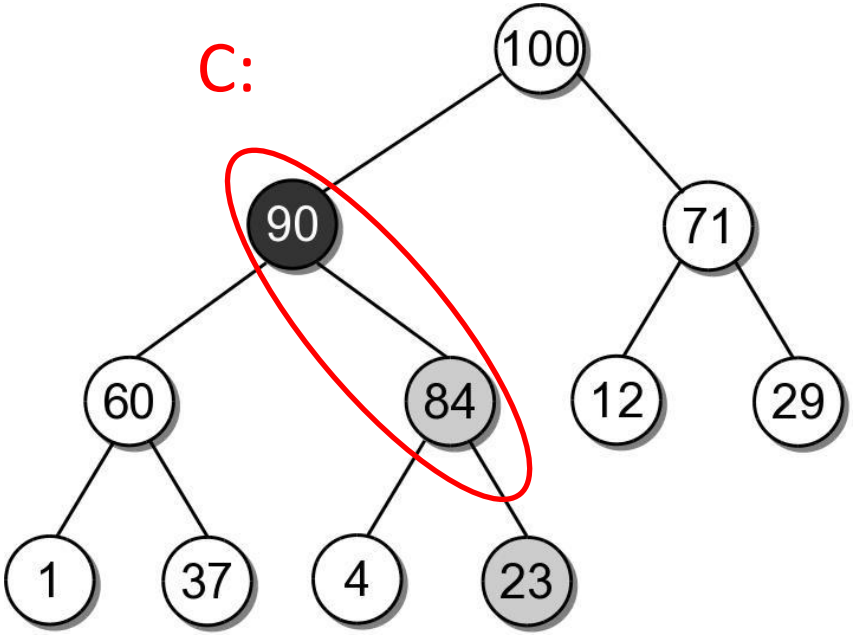
Insert

e.g. insert (90)

B: Since 90 is bigger than its parent, 23, swap them.

C: Now, 90 is bigger than its new parent, 84, swap again.





Heaps

Removing Items

e.g. removeMax()

- Remove root value, 100 , and eventually return this value.
- Where to get the new root?
  - Choose the last leaf (to maintain heap shape)
  - If result is not a heap, swap root value with larger child, and "sift down" until heap is identified

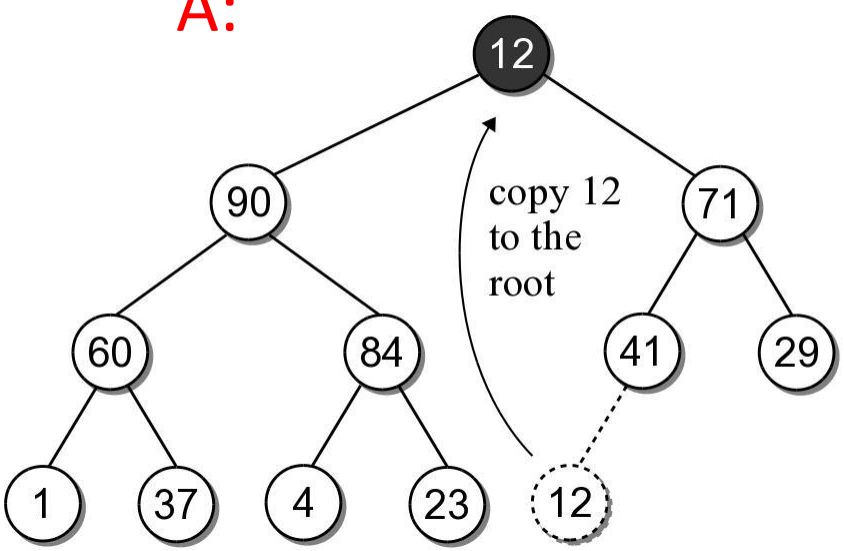
Heaps

Removing Items

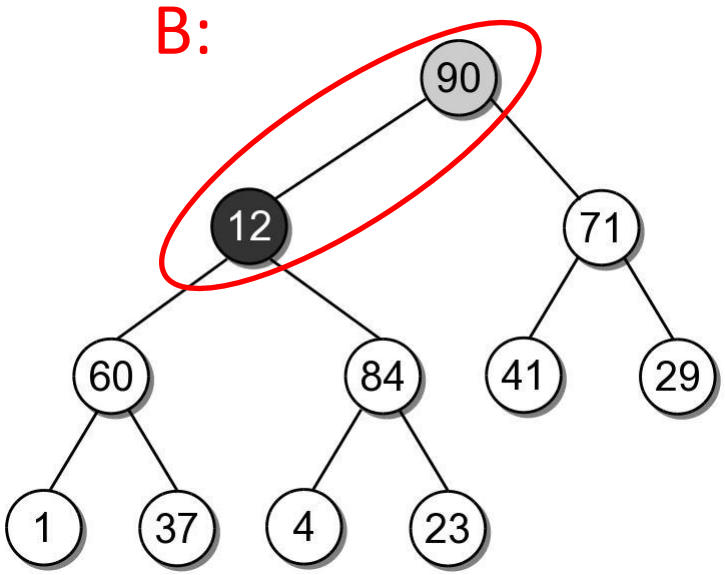
A: Replace root , 100, value with right-most leaf, 12

B: Swap 12 with max(90, 71). Look at 12's new children.

A:



B:

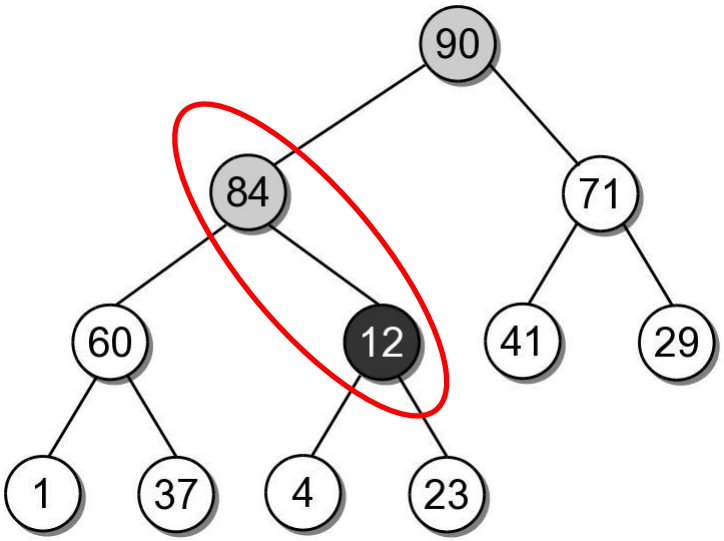


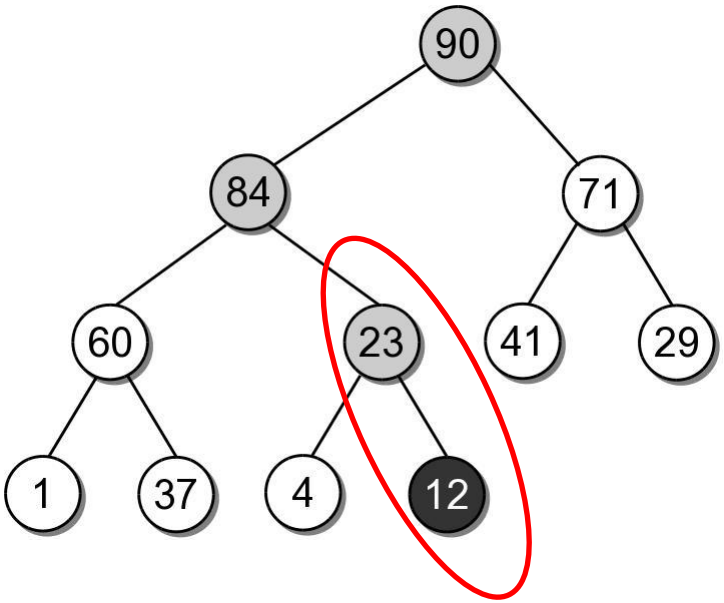
Heaps

Removing Items

A: Not a heap yet; swap 12 with max(4,23)

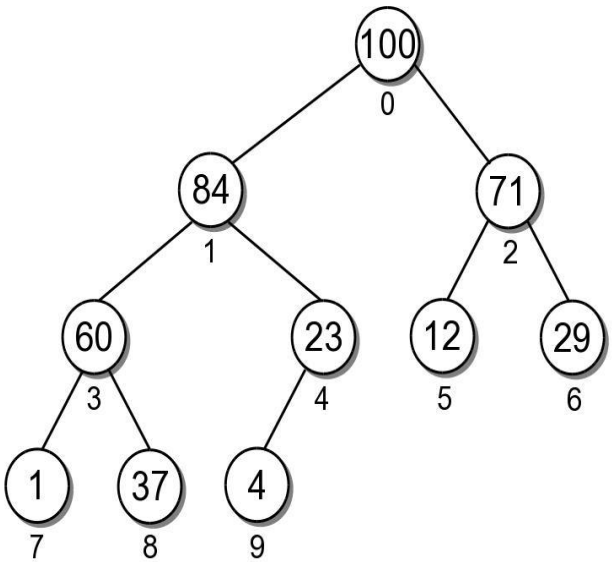
B: Finally, we have a heap!





Storing a Heap

Store heap using an array.



100	84	71	60	23	12	29	1	37	4
0	1	2	3	4	5	6	7	8	9

- Place root at index 0
- Place items into the array using breadth-first ordering
- For a node in the ith position?
  - Left Child: $2i + 1$
  - Right Child: $2i + 2$
  - Parent: $i - 1 // 2$

Heap Based Priority Queue

Heap Based Priority Queue

- Can implement an Unbounded Priority Queue using a max-heap
- Store items with their priorities
- Use priorities to shape the heap

Heap Based Priority Queue

Implementation

- **`L`** – use a Python list to store items with priorities
- **`PriorityQueue()`** constructor creates empty list

```
L = []
```

- **`isEmpty()`**

```
return len(L) == 0
```

- **`len()`**

```
return len(L)
```

Heap Based Priority Queue

Implementation of Enqueue

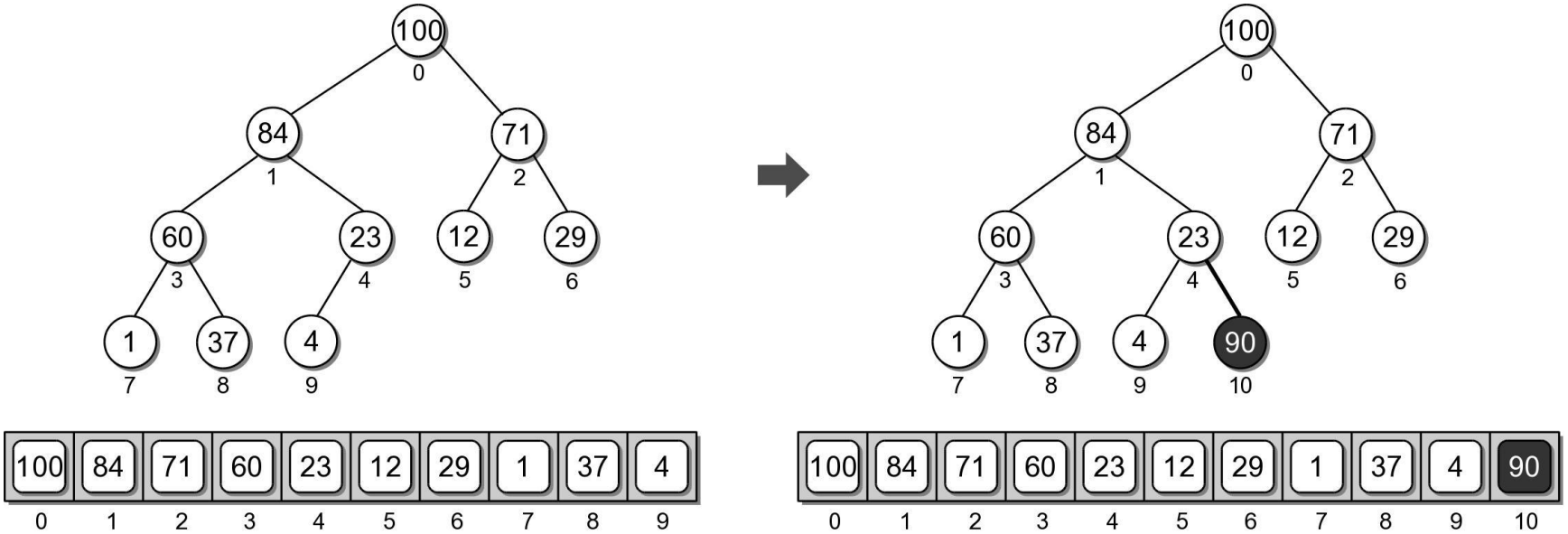
`pq.enqueue(item, priority)`

- Add item with priority to end of **`L`**
- while an item's priority > parent's priority:
  - swap item and parent
  - update indices for parent and child

Heap Based Priority Queue

Enqueue(90)

What actually happens when 90 is added to the heap?

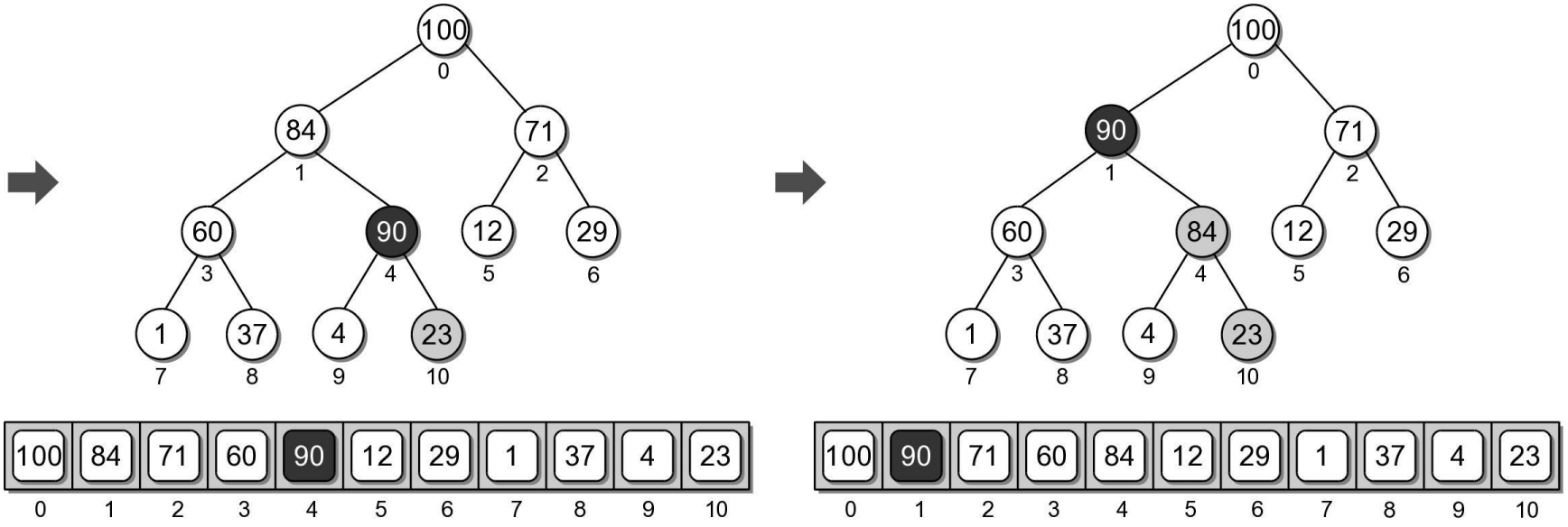


90 is added to the leftmost postion in the lowest level.

Heap Based Priority Queue

Enqueue(90)

The item 90 moves up the heap on edge at a time.



If 90 is larger than its parent, the two change positions.

Heap Based Priority Queue

Efficiency of enqueue

- *Adding* to end of heap:

  Python List: Worst case O(n) Amortized cost O(1)

- *Sifting* up –

Maximum iterations	O(log n)
Steps per iterations	O(1)
Total	O(log n)

- *Overall*:

  Worst case = O(n) Amortized cost = O(log n)

Heap Based Priority Queue

Implementation of Dequeue

- **`pq.dequeue()`**
  return root (i.e. item with maximum priority)
- Move last entry into root
- While parent's priority < max children's priority:
  - swap parent and child with higher priority
  - update indices for parent and child

Heap Based Priority Queue

Efficiency of dequeue

- *Removing*/Swapping last and first of heap:
  Python List: worst case O(n) amortized cost O(1)

- *Sifting* down

Maximum iterations	O(log n)
Steps per iterations	O(1)
Total	O(log n)

- *Overall*:
  worst case= O(n) amortized cost=O(log n)

Topic 11 – Sorting

Key Concepts

- Section 13.5 Heap Sort
- Section 5.2: Bubble Sort, Selection Sort, Insertion Sort
- Section 12.1 Merge Sort
- Section 12.2 Quick Sort
- Section 12.4 Radix Sort
- Comparing Sorts and Hybrid Sorts

Heaps

Another application of heaps

Suppose we have a list L.

- Create an empty heap
- Add each item in L to the heap
- Then remove max from heap and append to new list

➔ *We have a sorting algorithm!*

Heapsort

Asymptotic Time Complexity of Heapsort

- *Allocate* array for heap O(*n*)
- *Adding* to the heap:

  to add an item is at worst O(log *n*)

  n items are added

  total = O(n log n)
- *Removing* from heap / appending to list

  to remove an item is at worst O(log n)

  n elements removed

  total = O(n log n)

Improvements to Heapsort

In-place Sorting

- *Observation*: heapsort uses an additional array for the heap
- *Key Idea*: Use the original Python list instead
- Called an *In-place* algorithm

  Requires O(1) additional memory
- Have L share its space with the heap when

  - constructing the heap,

  - re-filling the Python list

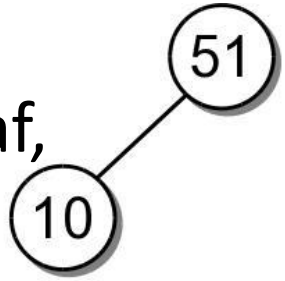
  - Heap always uses the left part of the Python list

In-place Heapsort

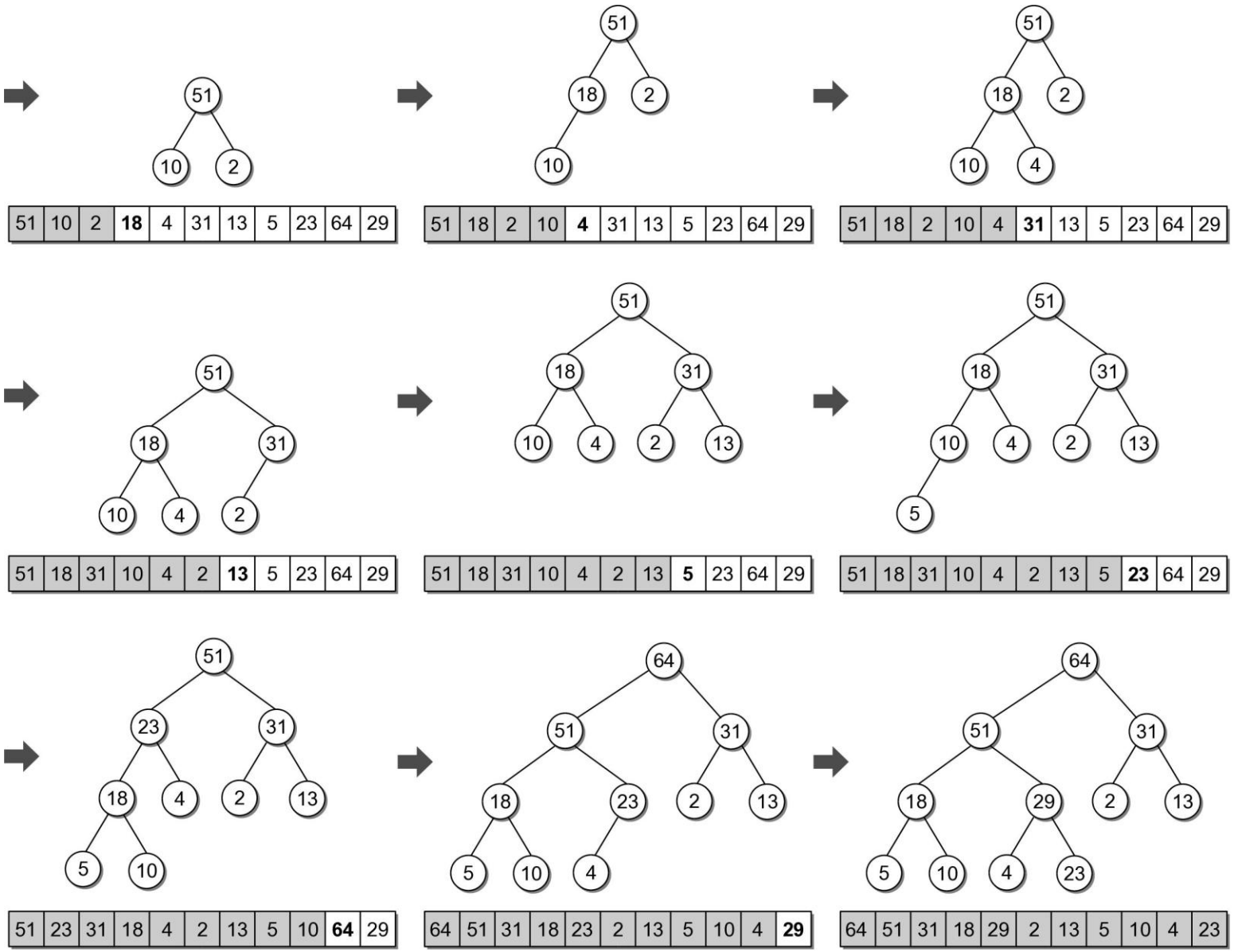
- Items in grey cells are in the heap.
- Insert the first item, 10, into the heap:

10	**51**	2	18	4	31	13	5	23	64	29

- Insert the 2nd item, 51, as new leaf, and then sift up



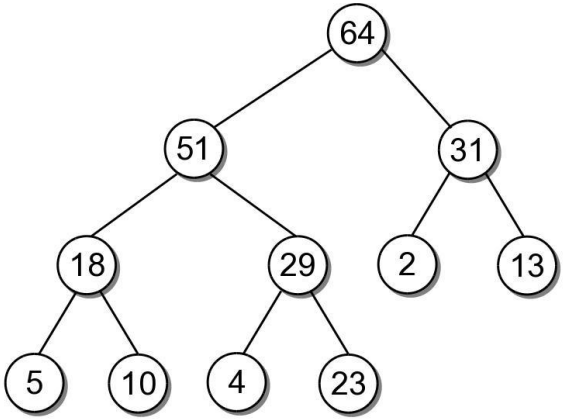
51	10	**2**	18	4	31	13	5	23	64	29


51 10 2 18 4 31 13 5 23 64 29
51 18 2 10 4 31 13 5 23 64 29
51 18 2 10 4 31 13 5 23 64 29
51 18 31 10 4 2 13 5 23 64 29
51 18 31 10 4 2 13 5 23 64 29
51 18 31 10 4 2 13 5 23 64 29
51 23 31 18 4 2 13 5 10 64 29
64 51 31 18 23 2 13 5 10 4 29
64 51 31 18 29 2 13 5 10 4 23

In-place Heapsort

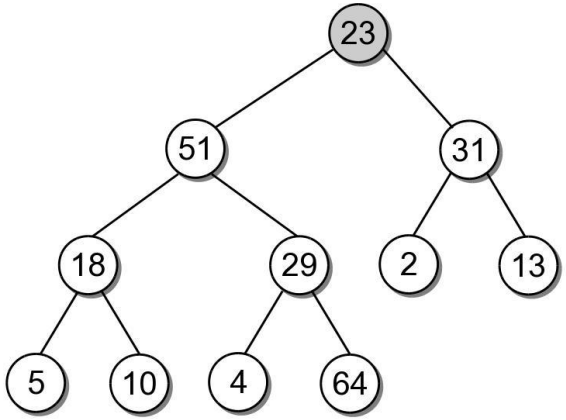
Retrieving the sorted data

- Repeatedly remove the max from heap
- Place it just past end of heap in the Python list
- Repeat until the heap is empty
- Array is now in sorted order!



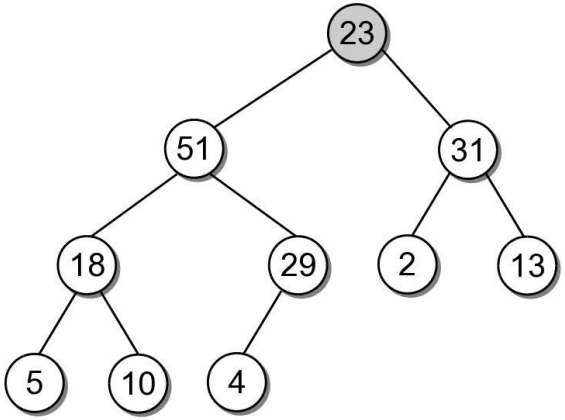
64	51	31	18	29	2	13	5	10	4	23

(a) the original max-heap.



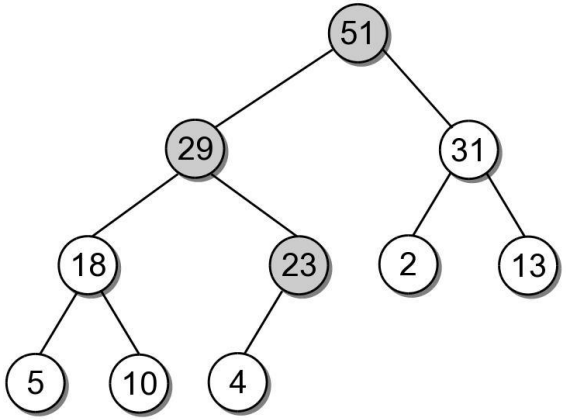
23	51	31	18	29	2	13	5	10	4	**64**

(b) swap the first and last items in the heap.



23	51	31	18	29	2	13	5	10	4	64

(c) remove the last item from the heap.



51	**29**	31	18	**23**	2	13	5	10	4	64

(d) sift the root value down the tree.

Basics of Sorting

- $\mathbf{L = [x_0, x_1, x_2, \ldots, x_n]}$
- Let L be a collection of comparable data
- By *comparable* we mean:
  - Given two unequal values, can determine which is bigger and which is smaller
- For simplicity, for theses slides we will always sort into *ascending order*
- For this topic: *n* is always the *number of items* to be sorted.

Desirable Properties

- *Comparisons of keys*: O(n log n) worst case
- *Swaps*: O(n) worst case
- *In place*: Requires O(1) additional memory
- *Stable*: Equal keys are not reordered
- *Special Case Performance ('adaptive')*: Asymptotic time complexity approaches O(n) for nearly sorted data.

A few basic sorts (to start)

- Bubble sort
- Selection sort
- Linear Insertion sort

http://www.sorting-algorithms.com/
http://www.cs.uwaterloo.ca/~bwbecker/sortingDemo/
http://math.hws.edu/TMCM/java/xSortLab/

Bubble Sort

Work from left to right

Compare pairs, if left is bigger than right then swap

Bubble up the largest item to the end.

Bubble up the 2nd largest item to 2nd from the end, etc

First bubble up the largest value, 9, to the last position.

8 5 9 3	8 > 5 swap
5 8 9 3	8 < 9 ok
5 8 9 3	9 > 3 swap
5 8 3 9	the largest item is now in the last position
	continue with just the first 3 values

Bubble Sort

Bubble up the 2nd largest value, 8, to the 2nd last position

5 8 3 9	5 < 8 ok
5 8 3 9	8 > 3 swap
5 3 8 9	the 2nd largest item is in the 2nd last position, continue with first 2 items

Bubble up the 3rd largest value, 5, to the 3rd last position

5 3 8 9	5 > 3 swap
3 5 8 9	Done

Selection Sort

Select the 1st smallest item, put it in the 1st position.

Select the 2nd smallest item, put it in the 2nd position.

Select the 3rd smallest item, put it in the 3rd position...

		8	5	9	3	6	*select* 3 (the smallest) put 1st
3			8	5	9	6	*select* 5 (the 2nd smallest) put 2nd
3	5			8	9	6	*select* 6 (the 3rd smallest) put 3rd
3	5	6			8	9	*select* 8 (the 4th smallest) put 4th
3	5	6	8			9	*select* 9 (the 5th smallest) put 5th
3	5	6	8	9			Done

Insertion Sort

1st item on right, *insert* in sorted position on left.

next item on right, *insert* in sorted position on left.

next item on right, *insert* in sorted position on left ...

	8	5	9	3	6
8		5	9	3	6
5	8		9	3	6
5	8	9		3	6
3	5	8	9		6
3	6	5	8	9	

insert 8 on left

insert 5 in sorted position on left

insert 9 in sorted position on left

insert 3 in sorted position on left

insert 6 in sorted position on left

Done

Mergesort and Quicksort

Divide and Conquer Approach

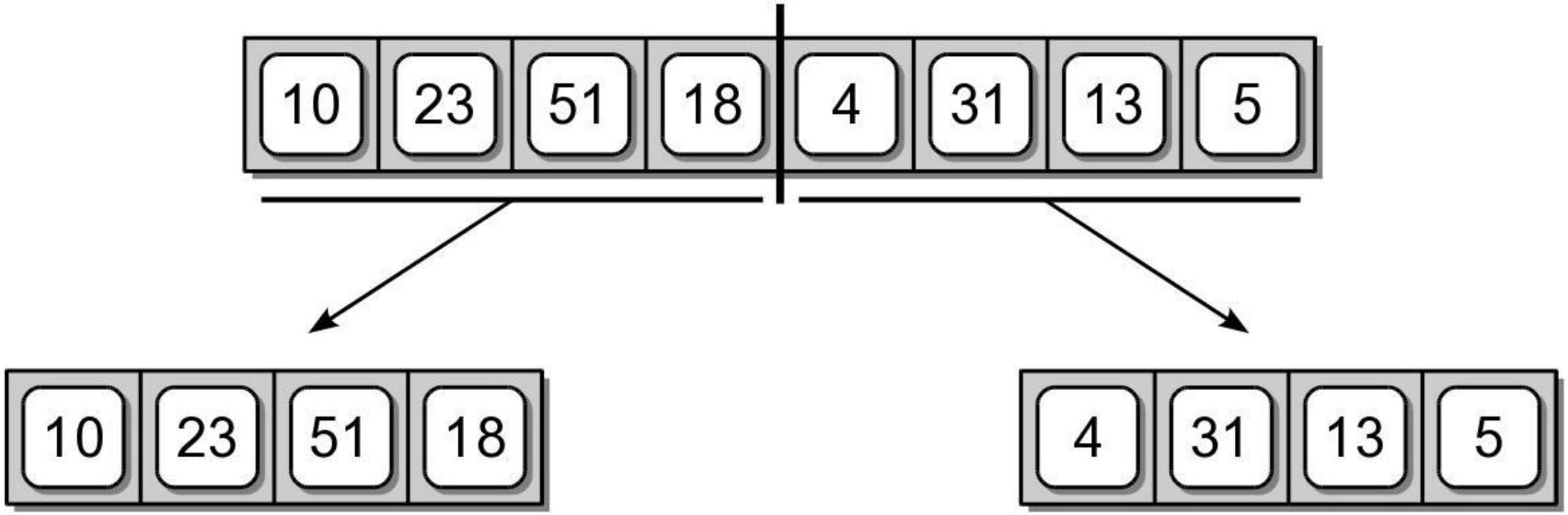
- Recursively *divide* the list into 2 halves
- *Conquer* (i.e. sort) the halves.
- *Combine* the sorted halves into one sorted list.

Examples

- Mergesort
- Quicksort

Mergesort and Quicksort

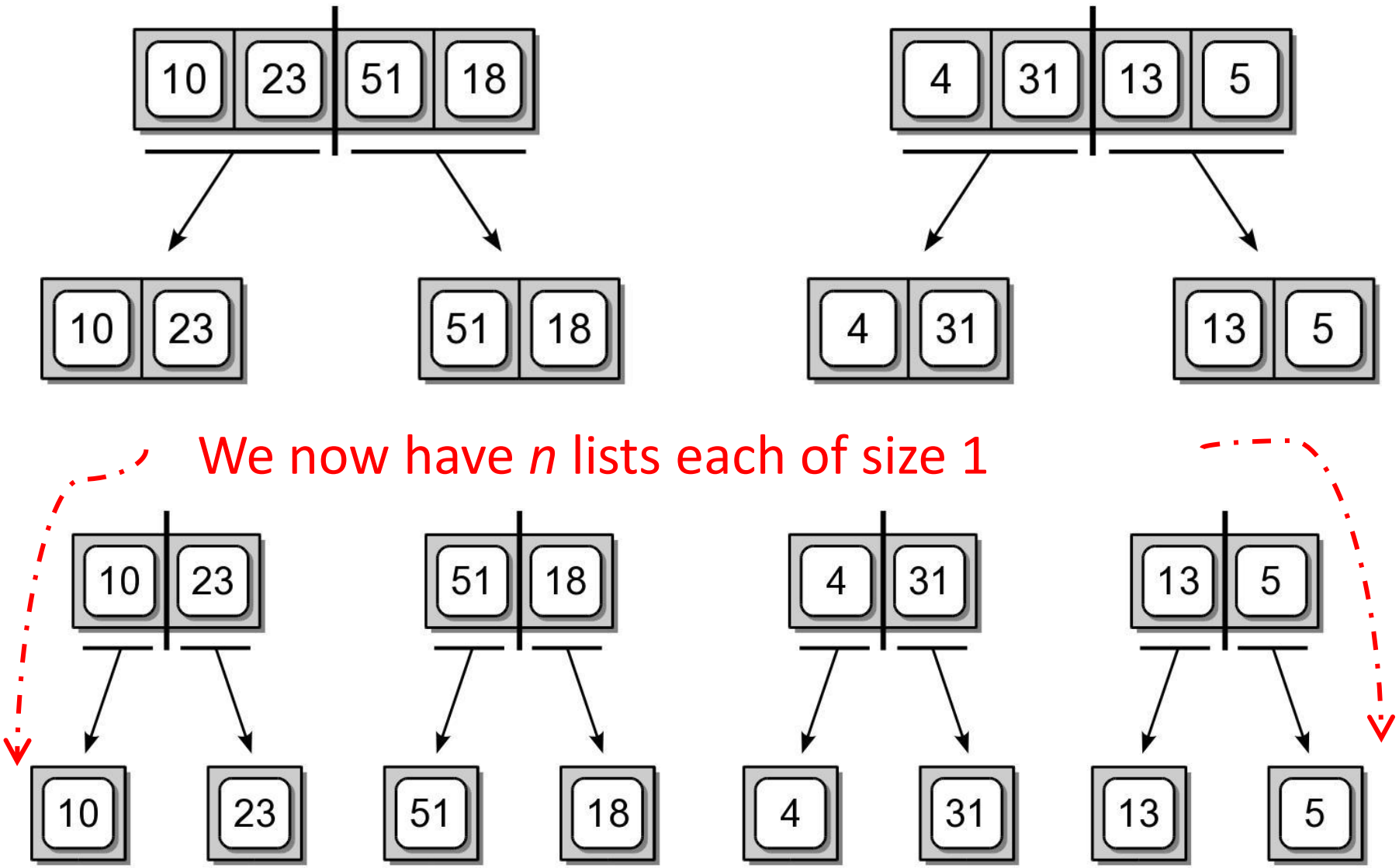
Divide the list in two:



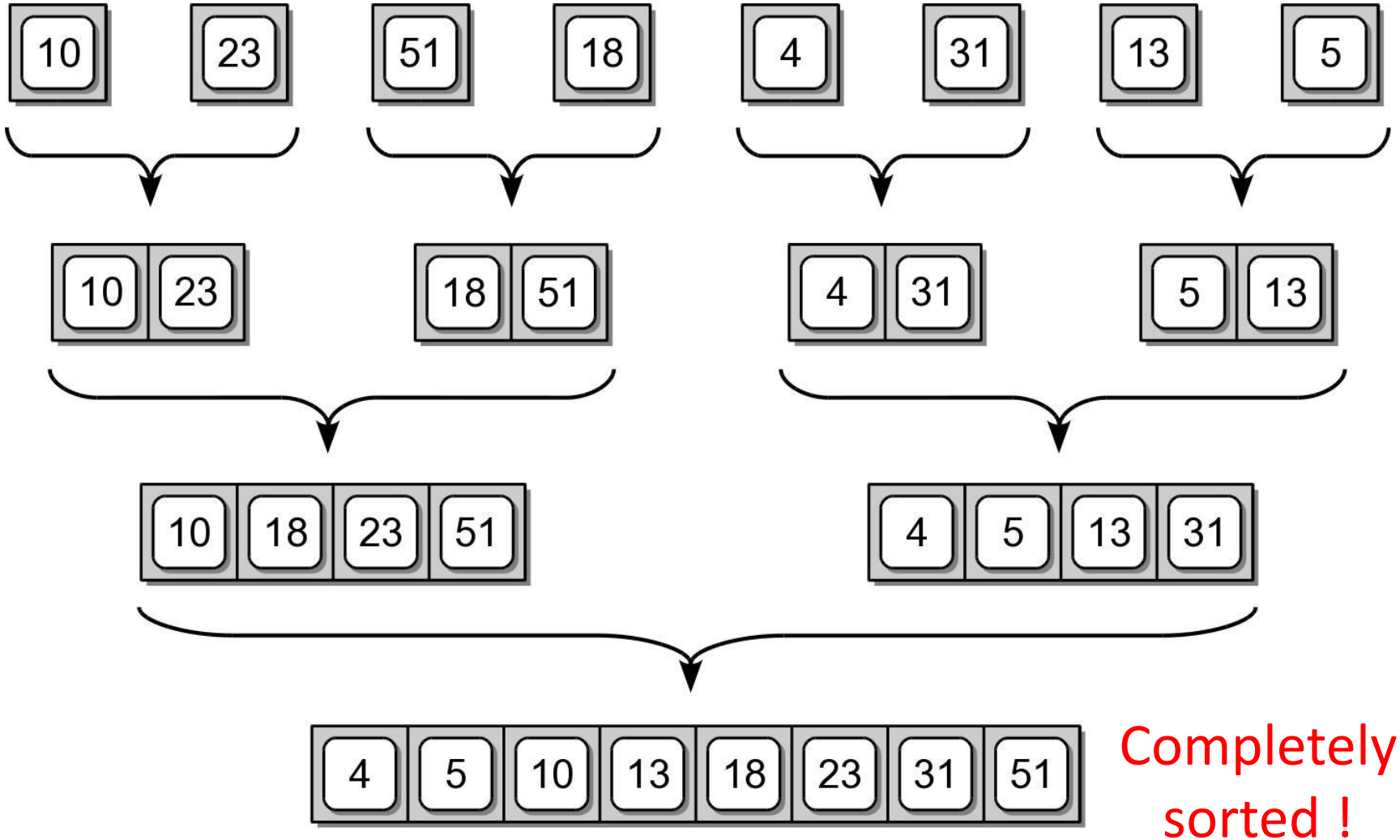

```
lhs = L[0:len(L)//2]
```

```
rhs = L[len(L)//2:]
```

Recursive divide over and over again



Merge the lists together, keeping them sorted.



Completely sorted !

Mergesort

Merging sorted lists A and B

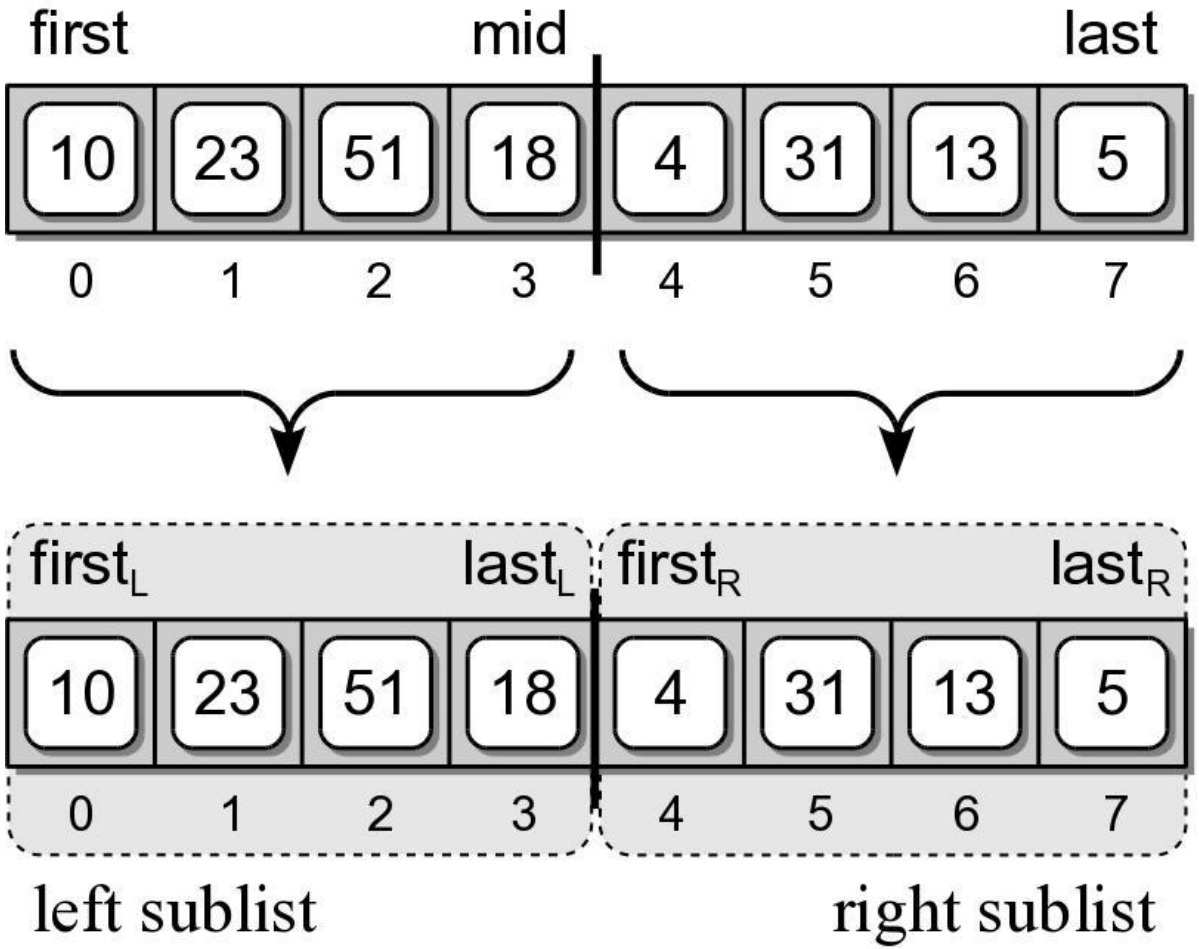
```
sorted = []
posA = 0, posB = 0
while posA < len(A) and posB < len(B):
   if A[posA] <= B[posB]:
      sorted.append(A[posA]), posA += 1
   else:
      sorted.append(B[posB]), posB += 1
if posA == len(A):
  sorted.extend(B[posB:])
else:
  sorted.extend(A[posA:])
```

Mergesort

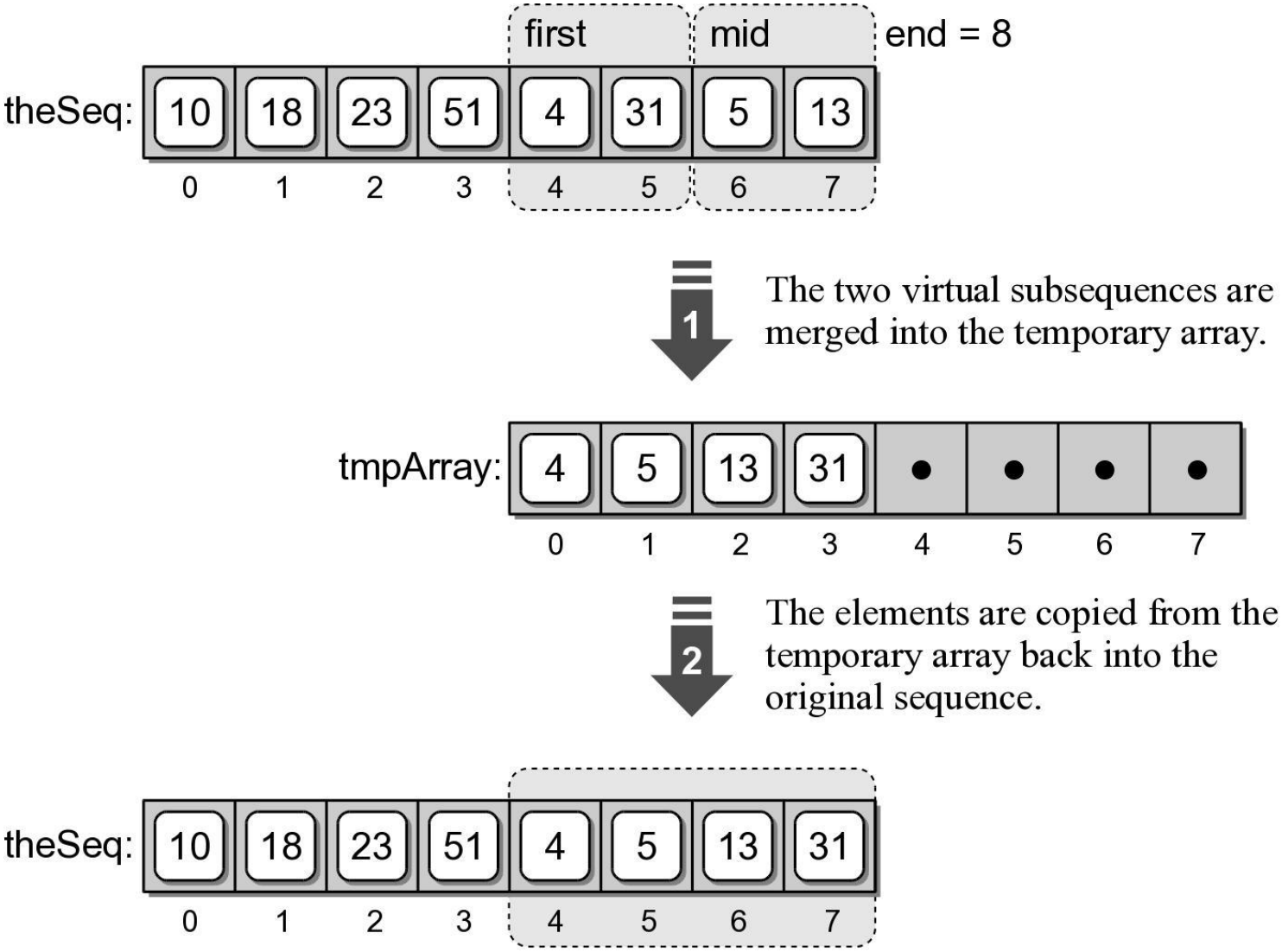
Problem

- Creates new lists with each division
- Additional memory needed for each recursive call
- Extra memory can be quite costly and can slow our algorithm down
- Instead
  - Allocate one temporary array
  - Merge everything into it
  - Use array indices to handle divisions

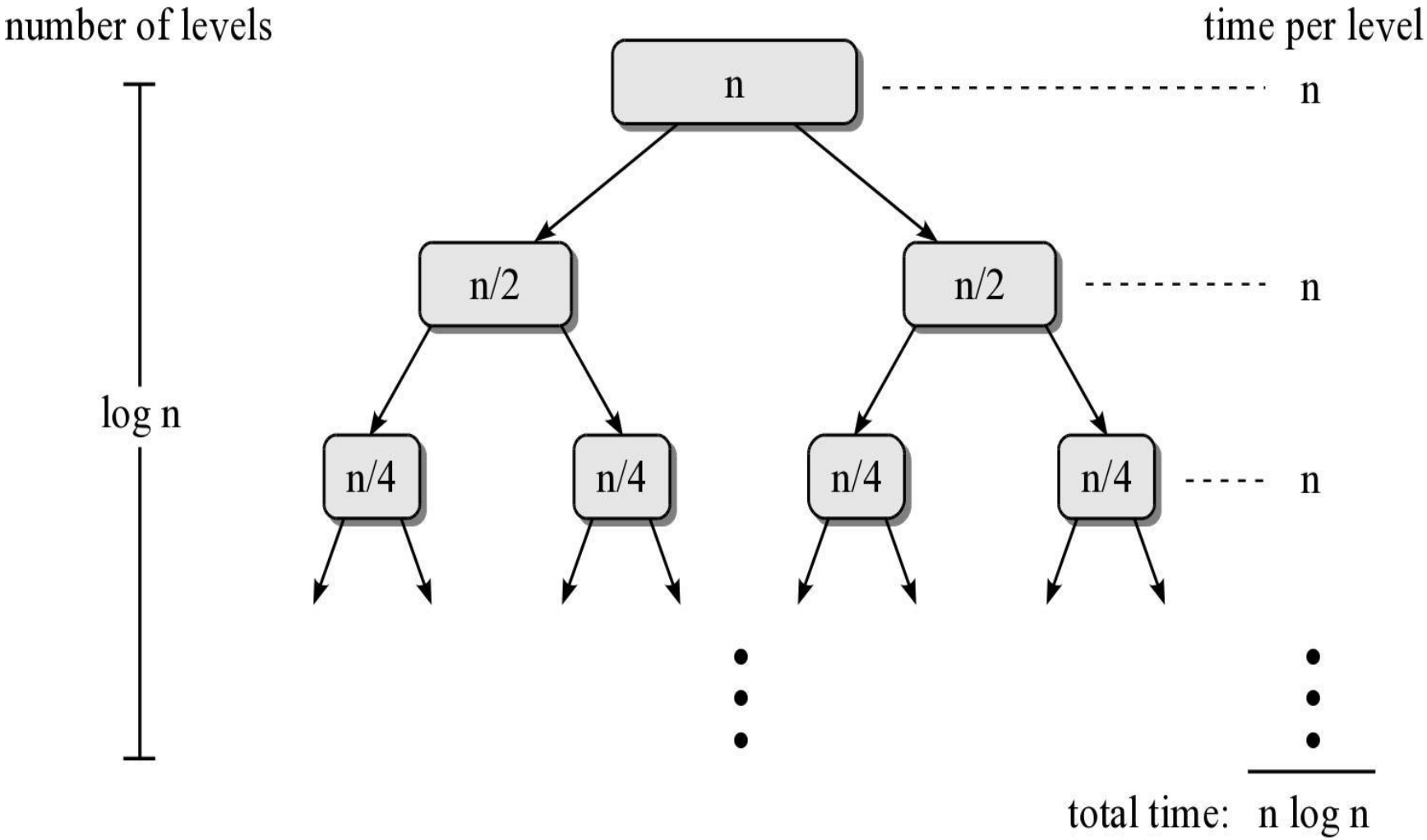
Managing the array: dividing



Managing the array: merging



Runtime Analysis



Quicksort

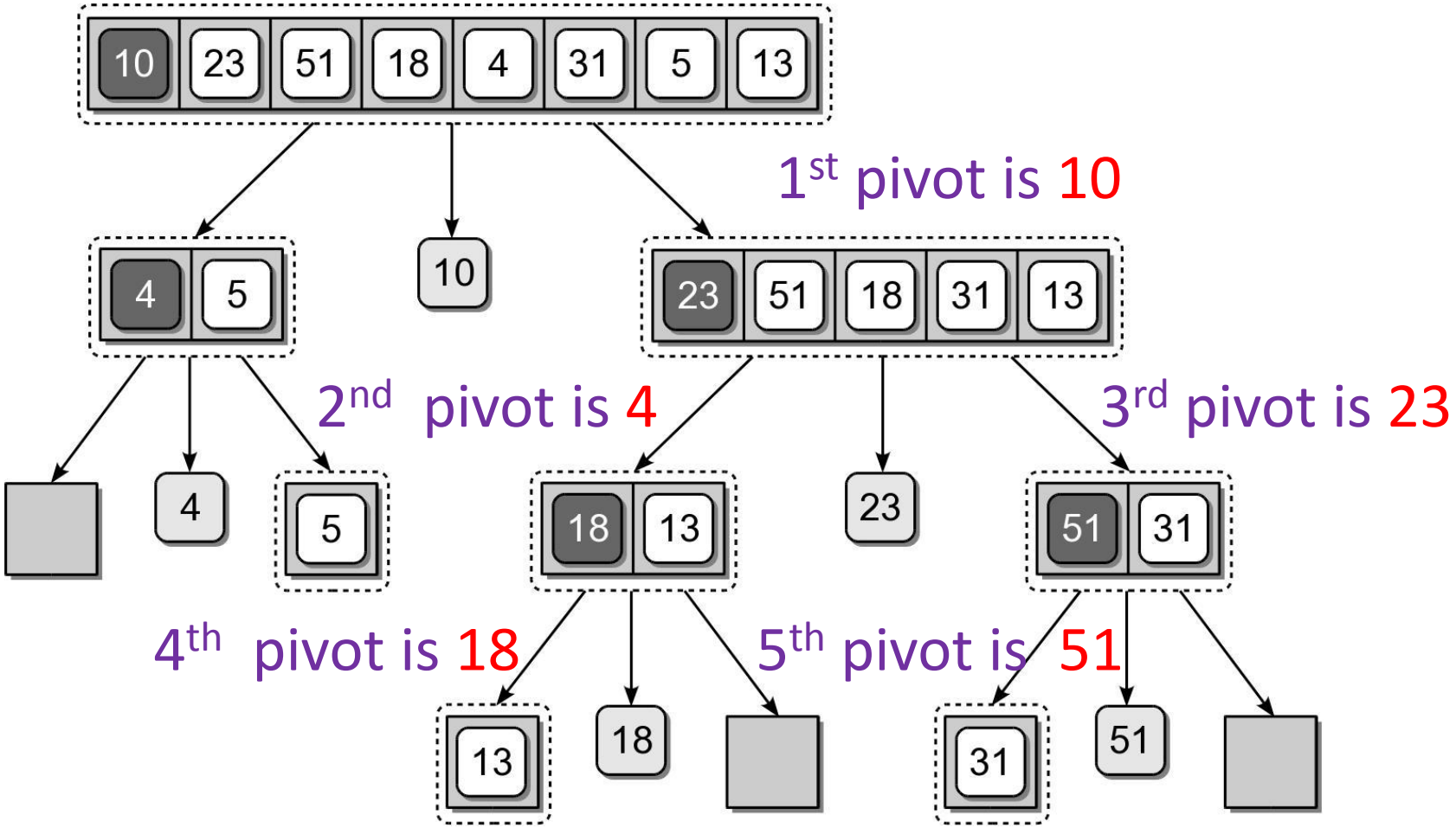
Basic Idea

- *Divide* and *conquer* but...
- Don't *divide* in half, instead *choose a pivot* from the list, and *divide* based on *pivot*
- left hand side (*lhs*) contains all the values < pivot
- right hand side (*rhs*) contains all the values ≥ pivot
- Sorted list is

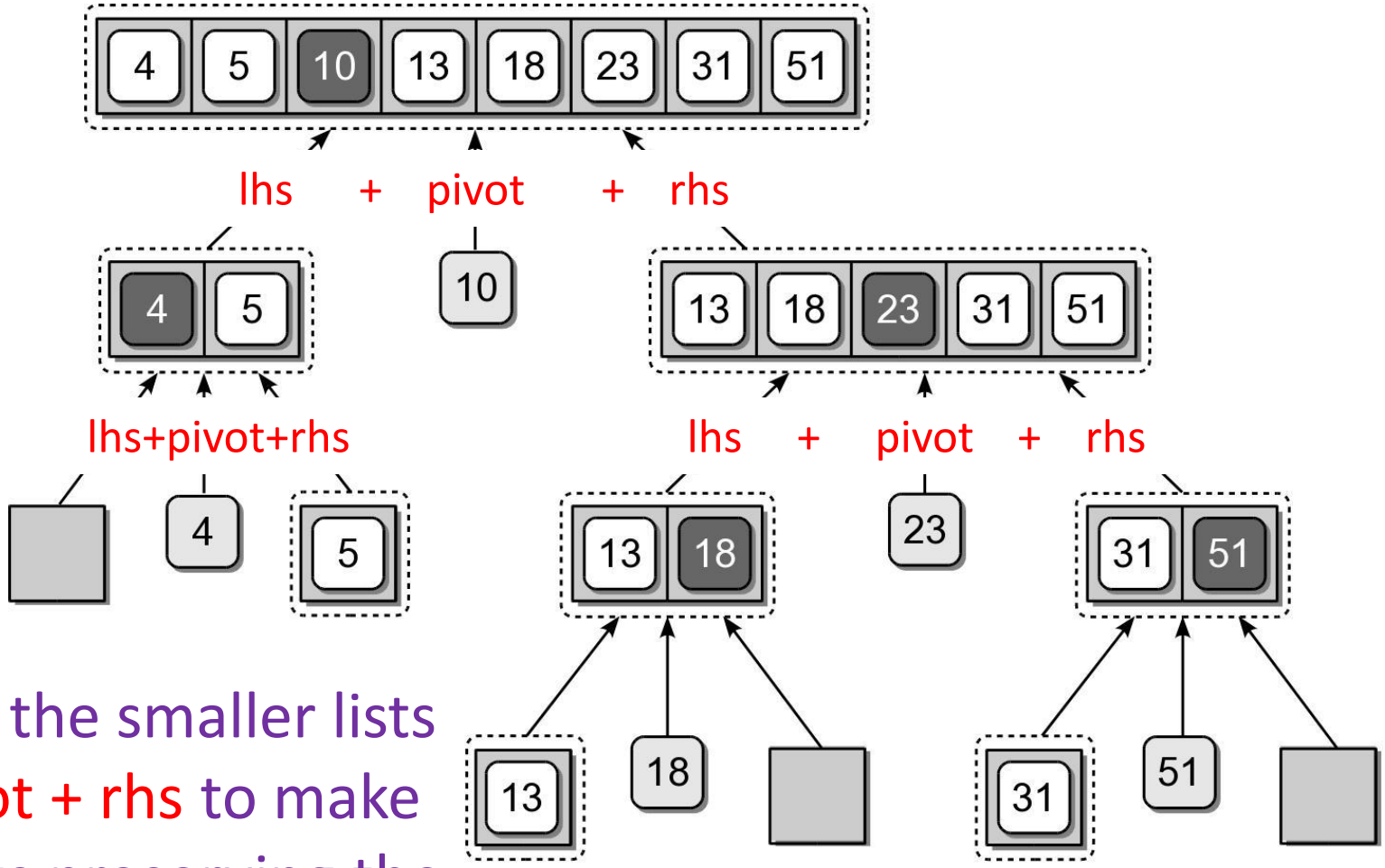
  Quicksort(*lhs*) + pivot + Quicksort(*rhs*)
- In Quicksort *rearranging* is done *when dividing*, in Mergesort *rearranging* is done *when combining*.

Quicksort: Divide list based on pivots

Basic Idea



Quicksort: Put sorted pieces back together



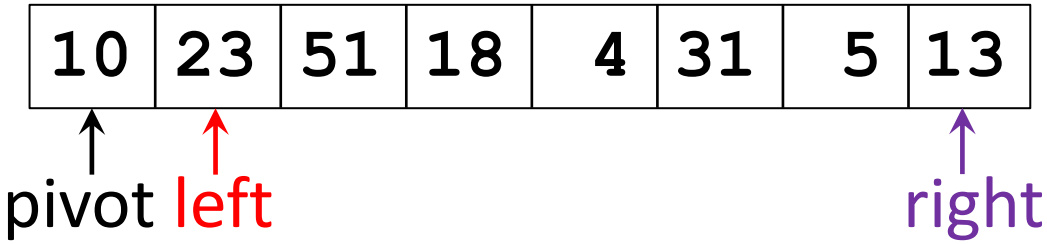
Combine the smaller lists lhs + pivot + rhs to make larger lists preserving the sorted order.

In-place Quicksort

Quicksort Variation

- partitioning is done in-place
- avoid the use of a temporary array
- given an array a[first : last+1]

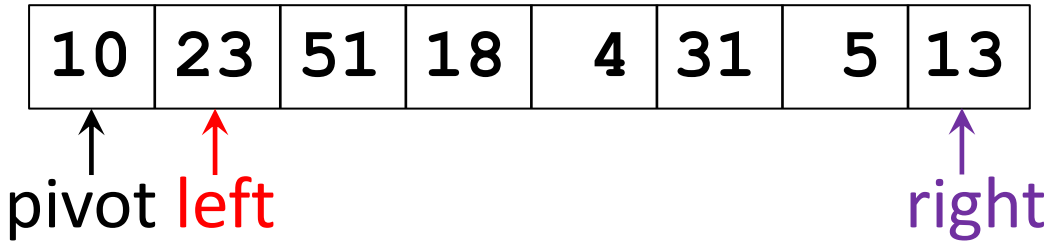
```
pivot = a[first]
left = first +1
right = last
```



In-place Quicksort

Find an item on the *left* to swap (i.e. 23)

```
# advance left until you find a big item
while left < right and a[left] < pivot:
    left += 1
```

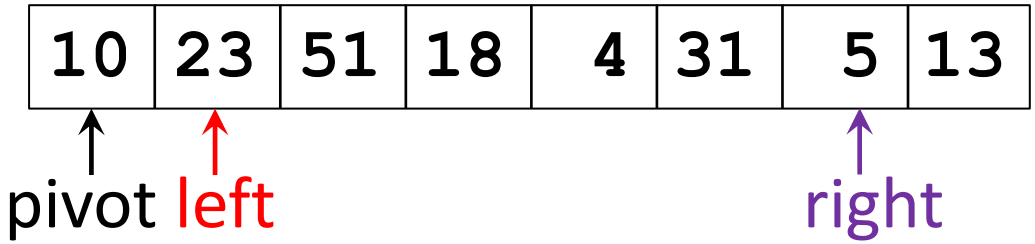


left is now pointing to an item (23) that is *greater than* the pivot (10)

In-place Quicksort

Find an item on the right to swap (i.e. 5)

```
# advance right until you find a small item
while left < right and a[right] >= pivot:
    right -= 1
```

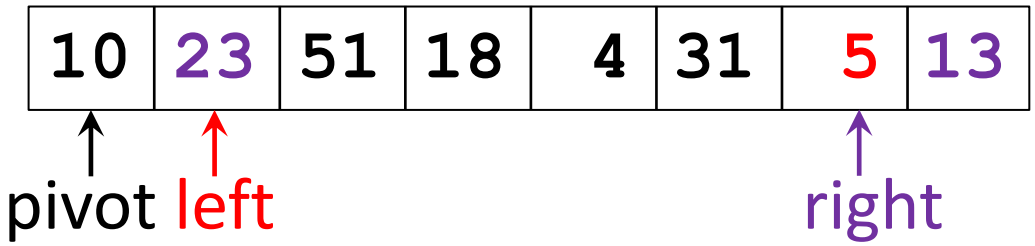


right is now pointing to an item (5) that is *less than* the pivot (10)

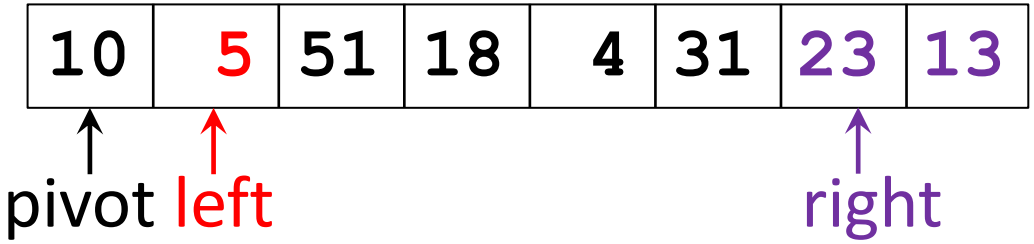
In-place Quicksort

Swap items

```
# swap the values at left and right
temp = a[left]
a[left] = a[right]
a[right] = temp
```



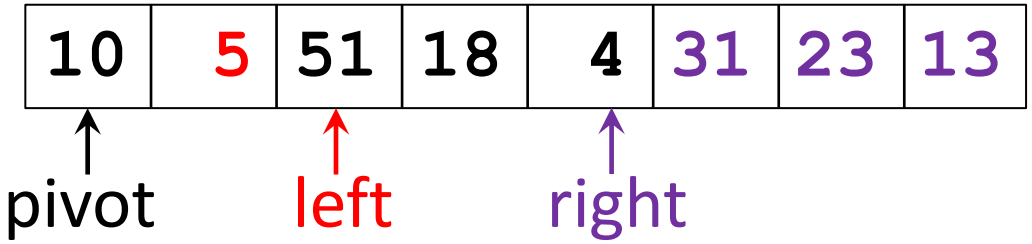
- Swap the location of items 5 and 23.



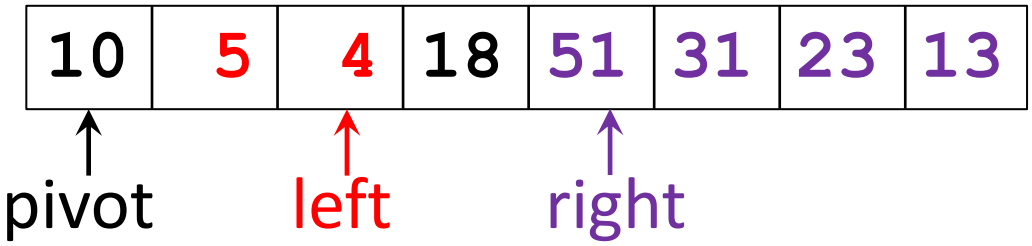
In-place Quicksort

Swap items

- Continue advancing the left until you find the next item *bigger* than the pivot, 51.
- Then, continue shifting the right until you find the next item *smaller* than the pivot, 4.



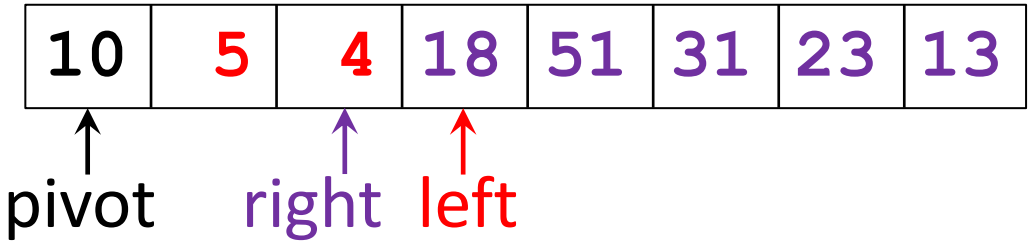
- Swap the location of items 4 and 51.



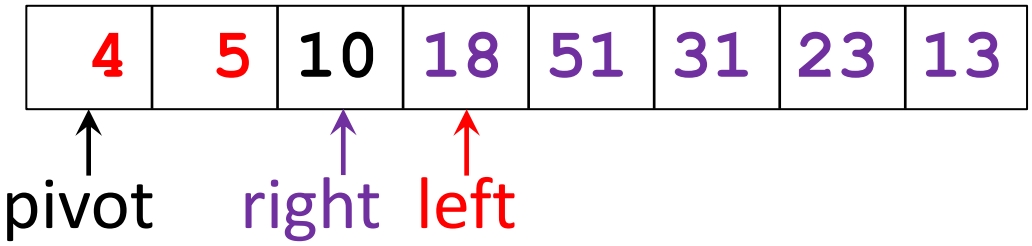
In-place Quicksort

Swap items

- Continue advancing the left until you find the next item *bigger* than the pivot, 18.
- Continue advancing the right until you find the next item *smaller* than the pivot or left > right.



- Swap the pivot (10) with what right is pointing to (4)



In-place Quicksort

One Recursive Step Complete

- You now have all the items less than the pivot on the left hand side (lhs) of the pivot.
- You now have all the items greater than the pivot on the right hand side (rhs) of the pivot.

4	5	10	18	51	31	23	13

lhs pivot rhs

- Call Quicksort on the lhs and the rhs.

Textbook Quicksort Error

There is an error in the textbook.

- The text starts off by sorting this array on page 351,

10	5	51	18	4	31	23	13
	left						right

- but then suddenly changes to this array on page 352,

10	2	21	13	5	81	48	18
	left						right

- and then changes back to the original array values on the bottom of page 352.

Efficiency of Quicksort

Worst case: O(n^2)

- pivot is always the smallest value, i.e. splitting into *lhs* with 0 items + *rhs* with n-1 items,
- for on iteration of n items it takes n-1 comparisons
- in total takes (n-1) + (n-2) + (n-3) + ... 2 + 1 = O(n^2) comparisons
- i.e. sum of an arithmetic series

Efficiency of Quicksort

Average case: O(*n* log *n*)

about half the time the pivot is close to the middle,

- *lhs* has at least $n/4$ values
- *rhs* has at least $n/4$ values
- the largest half is $< 3n/4$
- it takes $O(\log n)$ levels to get to arrays of size 2
- doing $O(n)$ comparisons at each level
- total $O(n \log n)$

Radix Sort

Bucket Sorts: Non-comparison Based Sorting

- Given aList containing values in the range 0 .. *M*, for some *M* > 0
- Create *M*+1 *buckets* and track the number of items

```
counts = [0]*(M+1)
for value in aList:
    counts[value] += 1
```

- Create the sorted list, in order, from **`counts`**
- Time complexity: O(*n* + *M*)
- Good when there is a small range of values (i.e. 0 .. M)

Radix Sort

General Bucket Sort

- Create *k* buckets (e.g. for ranges of values)
- Place each item in appropriate bucket
- Sort each bucket individually
- Re-combine in sorted order

Radix Sort

Radix Sort

- Task: sort aList of *n* items, each with ≤ *D* digits
- Place empty queues into 10 buckets (0..9)

```
for d in range(D, 0, -1):
  for each value:
    add to bucket based on dth digit
  for each bucket:
    transfer all values, in bucket order, to list
```

- For my example assume only two digits, 0 and 1

Radix Sort

Radix Sort

input: 110 010 001 101 011 100

d=3, place in bucket based on 3rd digit

bucket 0: 110 010 100

bucket 1: 001 101 011

place back in list, starting with bucket 1 then bucket 0

list: 001 101 011 110 010 100

Radix Sort

Radix Sort

list: 001 101 011 110 010 100

d=2, place in bucket based on 2nd digit

bucket 0: 001 101 100

bucket 1: 011 110 010

place back in list, starting with bucket 1 then bucket 0

list: 011 110 010 001 101 100

Radix Sort

Radix Sort

list: 011 110 010 001 101 100

d=1, place in bucket based on 1st digit

bucket 0: 011 010 001

bucket 1: 110 101 100

place back in list, starting with bucket 1 then bucket 0

list: 110 101 100 011 010 001

Radix Sort

Efficiency of Radix Sort

- Choose CLL implementation of queue
  O(1) enqueue, O(1) dequeue
- Create 10 empty queues: O(1)
- For each digit, i.e. O(d) iterations
  Place each value into a queue: O(n)
  Empty queues into list: O(n)
  → O(dn)

Note: if k buckets → O($k + dn$)

Comparing Sorts

Quick Review

- So far, we've seen

  Bubble sort, Selection sort, Insertion sort

  Heapsort, Mergesort, Quicksort, Radix sort
- Comparison-based sorting has $\Omega(n \log n)$ complexity
- Stable algorithms useful for

  - priority queues

  - sorting bases on multiple columns of data

Comparing Sorts

Downside of Divide and Conquer

- When a function is called in recursion, an "activation" record is pushed onto a stack
- The amount of memory required due to recursion corresponds to

  - the maximum length of the stack

  - the maximum depth of the "recursion tree"
- Both *Mergesort* and *Quicksort* can be implemented with iteration instead

Hybrid Algorithms

Insertion Sort

- Low overhead
- Works very well when data is close to sorted

Mergesort and Quicksort

- Relatively high overhead for small lists
- Works well overall

Key Idea

- Used *one sort* when *lists are large,* then switch to the *other sort* when *lists get small.*

Hybrid Algorithms

Python's sort

- Previously used *Quicksort*
- Now uses *Timsort* (named after Tim Peters)
- *Timsort* is a hybrid sort: *Mergesort* with *Insertion sort*
- O(n log n) comparisons
- Stable

Hybrid Algorithms

C++ std::sort

- No fixed standard

For large datasets it typically uses *introsort,* a hybrid of

- *Quicksort* initially
- *Heapsort* when list size gets smaller

For small datasets, it typically uses *Insertion sort*

So C++ std::sort is typically a combination of *three sorts!*
Quicksort + *Heapsort* + *Insertion sort*

Comparing Sorts

Which sorting technique is better?

It depends ….

- *Bubblesort* and *Selection sort* are easy to code
- *Insertion* sort is really fast for almost sorted data
- *Mergesort* is stable and always *O(n log n)*
- *Heapsort is O(n log n)*
- *Quicksort* is usually faster than *Mergesort*
- *Radix sort* is (kind of) *O(n)*

Comparing Sorts

Which sorting technique is better?

But ….

- *Bubblesort, Selection sort, Insertion sort,* and *Quicksort* are all $O(n^2)$ worst case
- *Mergesort's* fastest implementation requires additional memory
- *Heapsort* isn't *generally* as fast as *Quicksort*
- *Radix sort* requires specialized data

Comparing Sorts

An Underlying Assumption

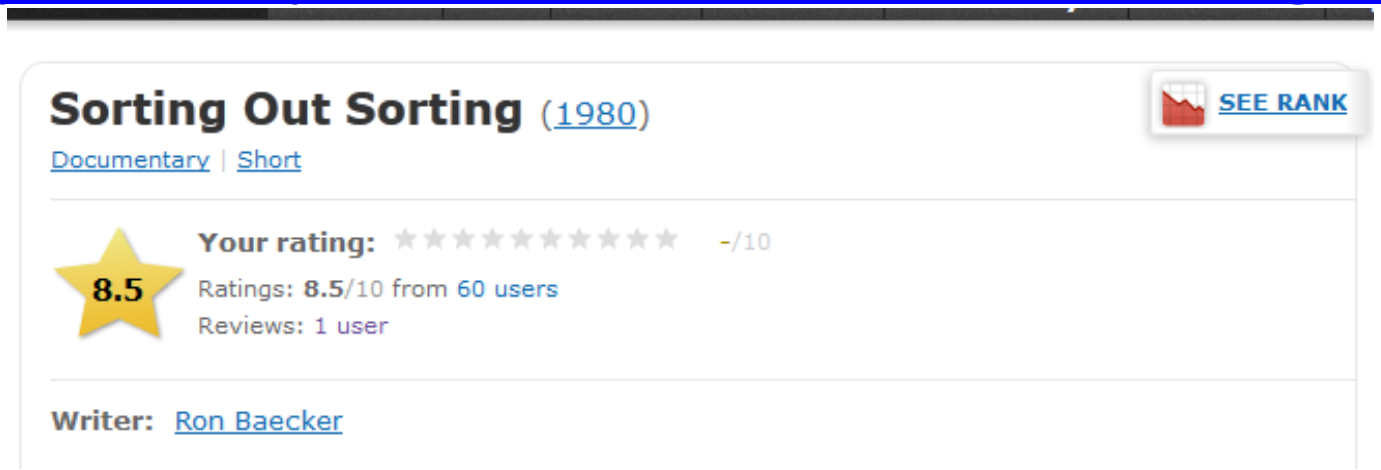
- All our discussions have assumed that all data is in main memory.
- Without that assumption, things are quite different.
- Using less memory is better for speed.
- Using data in order (i.e. not going to random spots in an array) is better for speed.

Comparing Sorts

Other Resources

- Lots of videos on youtube
- i.e. Hungarian Dancers doing Quicksort
- Also "Sorting out Sorting“

  http://www.youtube.com/watch?v=SJwEwA5gOkM



Topic 12 – Searching and Hashing

Key Concepts

- Section 5.1: Searching
- Section 11.1: Direct Access
- Section 11.2: Hashing
- Section 11.3 Separate Chaining
- Section 11.4: Hash Functions

Searching

Task: Find a record in a collection

- *More Specifically*: Searching for a *target key*
- Collection may be sorted or unsorted

Initial Assumptions (for simplicity):

- Data in an array-based collection, i.e O(1) access
- Only interested in key values
- *Keys* are integers
- *Keys* are not repeated
- E.g. student number / account number

Sequential (linear) Search

Key Idea: Start at the beginning of the list, and check each item until answer is known

- Array may be sorted or unsorted
- Simplest search

Asymptotic Time Complexity

- Best case: O(1)
- Worst case: O(n)
- Average case (successful search): $n/2$ i.e. O(n)
- All cases (unsuccessful search): n i.e. O(n)

Sequential (linear) Search

Key Idea: Start at the beginning of the list, and check each item until answer is known.

```
def LinearSearch_Unsorted(aKey, aList):
    for i in range(len(aList)):
        if aList[i] == aKey:
            return True
    return False
```

for code, see example18_Search.py in Learn

Sequential (linear) Search Sorted

Question

Can we speed up the search if the data is sorted?

Example

Consider searching for 9 in the list [1, 4, 7, 10, 28, 31]

- When can we conclude 9 is not there?

Asymptotic Time Complexity

- Best case: O(1)
- Worst case: O(n)
- Average case: O(n)

Sequential (linear) Search

Key Idea: Stop when you found a value larger than the key.

```
def LinearSearch_Sorted(aKey, aList):
    for i in range(len(aList)):
        if aList[i] == aKey:
            return True
        elif aList[i] > aKey:
            return False
    return False
```

for code, see example18_Search.py in Learn

Binary Search

Can we do better for sorted data?

Task: Search for **aKey** in **L** from **lo** to **hi**

Key Idea: Look at the middle entry, **L[mid]**

```
if aKey == L[mid]:
    return True
if aKey < L[mid]:
    search in left half  (i.e. from lo to mid-1)
if aKey > L[mid]:
    search in right half  (i.e. from mid+1 to hi)
```

Binary Search

```
def BinarySearch(aKey, aList):
    lo = 0
    hi = len(aList) -1

    while (lo <= hi):
        mid = (hi + lo) // 2
        if aList[mid] == aKey:
            return True
        elif aList[mid] < aKey:
            lo = mid + 1
        else:
            hi = mid - 1
    return False
```

for code, see example18_Search.py in Learn

Binary Search

Example 1a: Search for 121

```
key = 121
lo = 0
hi = 7
mid = (7+0)//2 = 3
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
lo			mid				hi

```
a[mid] > 121
hi = mid - 1 = 2
```

Binary Search

Example 1b: Search for 121

```
key = 121
lo = 0
hi = 2
mid = (2+0)//2 = 1
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
lo	mid	hi					

```
a[mid] < 121
lo = mid + 1 = 2
```

Binary Search

Example 1c: Search for 121

```
key = 121
lo = 2
hi = 2
mid = (2+2)//2 = 2
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
		lo hi mid					

```
a[mid] == 121
return True
```

Binary Search

Example 2a: Search for 160

```
key = 160
lo = 0
hi = 7
mid = (7+0)//2 = 3
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
lo			mid				hi

```
a[mid] < 160
lo = mid + 1 = 4
```

Binary Search

Example 2b: Search for 160

```
key = 160
lo = 4
hi = 7
mid = (7+4)//2 = 5
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
				lo	mid		hi

```
a[mid] < 160
lo = mid + 1 = 6
```

Binary Search

Example 2c: Search for 160

```
key = 160
lo = 6
hi = 7
mid = (7+6)//2 = 6
```

101	117	121	134	149	151	162	177
0	1	2	3	4	5	6	7
						lo	hi
						mid	

```
a[mid] > 160
hi = mid - 1 = 5  now hi < lo STOP!
```

Binary Search

Asymptotic Worst Case

- Maximum number of times aList can be divided in half

  $O(\log n)$
- Time for while loop:

  $O(1)$
- Worst case asymptotic time complexity:

  $O(\log n)$

Interpolation Search

- Observation: Binary Search always starts in the *middle* of the array
- Idea: Start where the key *would likely be located*
- Assume: values in array are uniformly distributed
- Example: Suppose you have 101 random integers between 0 and 1000 and you have sorted them and stored them in an array.
- Question: if the number 500 was one of the random integers, roughly where would you expect to find it?

Interpolation Search

Answer:

- 500 is in the middle of the random integers, which run from 0 to 1000.
- It would likely be in the middle of the array,
- The array runs from locations 0 to 100, so it would likely be near location 50, if it were there at all.
- Random Integers: 0 500? 1000

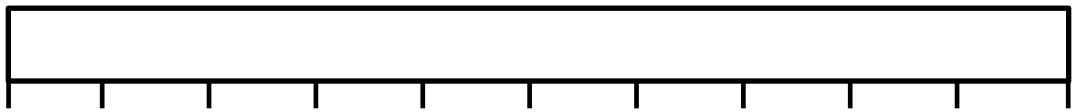
- Array Locations L 0 10 20 30 40 50 60 70 80 90 100

Interpolation Search

More examples

- Likewise, the number 200, if it was in the array at all, woud likely be near location 20.
- Likewise, the number 700 if it was in the array at all, woud likely be near location 70.

- Random Integers: 0 200? 500? 700? 1000



- Array Locations L 0 10 20 30 40 50 60 70 80 90 100

- *In words:* the ratio: *key / number range* is approximately equal to the ratio: *position/ array range*

Interpolation Search

More examples

- If random numbers and array start at 0 we have

  in this case: 700 / 1000 ≈ *pos* / 100

  generally: *key* / *num range* ≈ *pos* / *array range*

- If the random numbers range from L[*lo*] to L[*hi*] and the array ranges from *lo* to *hi* and we have
- $\underbrace{(key - L[lo]) / (L[hi] - L[lo])}_{\text{array entries}} \approx \underbrace{(pos - lo) / (hi - lo)}_{\text{array locations}}$

Interpolation Search

Task: Search for **key** in **L** from **lo** to **hi**

```
while lo <= hi:
    calculate pos
    if key == L[pos]:
        return True
    elif key < L[pos]:
        search from lo to pos-1
    else:
        search from pos+1 to hi
```

Efficiency of Interpolation Search

Average Case

- O(log log *n*)
- assume numbers are uniformly distributed
- won't prove this

Worst Case

- O(n)

Conclusion

- generally better than binary search
- but may be much worse

Searching Linked Lists

Sequential Searches (Slower)

- two methods: 1) sorted, 2) unsorted lists

Random Access Searches (Faster)

- two methods: 1) Binary Search 2) Interpolation Search

Do these faster searches *work for linked lists*?

- No, need O(1) random access to all locations
- Binary and Interpolation Search become linear, even if data is sorted.
- Don't bother sorting – it isn't worth it for linked lists

Minimizing Expected Retrieval Time

- *Observation*: Searching in a database
  - Typically, some values (around 20%) are retrieved very often.
  - Typically, many values (around 80%) are rarely retrieved.
  - Called the *80/20 rule*
- *Key Idea*: Concentrate on data that is frequently accessed → reduce expected retrieval time

Minimizing Expected Retrieval Time

- *Key Idea*: calculate T_{exp} the expected number of steps it will take to retrieve any item
- Let p_k be the probability that the user will search for the item at position k

  Note: $p_1 + p_2 + p_3 + \ldots + p_n \leq 1$

  may search for an item not in the list
- Start at position 1, then 2, then 3 ...
- It takes k steps to find the item in position k
- $T_{exp} = 1*p_1 + 2*p_2 + 3*p_3 + \ldots + n*p_n$

Minimizing Expected Retrieval Time

- If all *probabilities are equal*, i.e. $p_k = 1/n$
  then $T_{exp} = (1+2+3+\ldots+n)/n = (n+1)/2$

- If *probabilities are not all equal*:
  - to minimize the value of T_{exp} we need
  $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n$

- Therefore, rearrange the items from *most popular to least popular*

Minimizing Expected Retrieval Time

- Suppose we have a list of 4 courses, where

  ½ of all retrievals are for course 1

  ⅛ of all retrievals are for course 2

  ⅛ of all retrievals are for course 3

  ⅛ of all retrievals are for course 4

- For the order [course 4, course 3, course 2, course 1]

  $T_{exp} = 1 \cdot \frac{1}{8} + 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{2} = 2\frac{3}{4}$

- For the order [course 1, course 2, course 3, course 4]

  $T_{exp} = 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{8} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{8} = 1\frac{5}{8}$

How to approximate this order?

Approach 1: **Access Counts**

- *Key Idea: Maintain a count* of how many times each item has been retrieved in the past
- *Update count* when retrieved
- Allow just-retrieved value to move forward in the list if its count warrants it

  + Pros: Data is stored in optimal order *eventually*

  - Cons: It takes a long time to converge

  - Cons: Extra memory required to store count

  - Cons: Slow to respond to changes

How to approximate this order?

Approach 2: Move to Front

- *Key Idea: When an item is retrieved, move it to the front* of the list.
- Items move quickly

  + Infrequently accessed items move back

  + No need to maintain history

  + No need to maintain a count

  - *Not optimal* ordering, but generally performs pretty well

How to approximate this order?

Approach 3: Transposition

- *Key Idea: When an item is retrieved, swap it* with the element before it
- Frequently accessed elements will slowly make their way to the front
- Infrequently accessed elements won't generally make it to the front (unlike Move to Front)
- Pro: no need to maintain a count

Searching Unsorted Data

Does implementation matter?

Linked list

- Speed up average case by re-ordering data to reflect usage
- Worst case remains O(n), but actual retrievals usually faster

Python list/array

- Can use same ideas

Searching Sorted Data

Approach: Binary Search

- *at each step cut the size of the search space in half*
- a divide and conquer approach
- Pros: no tracking of counts, no swapping items
- Cons: data must be sorted

Can we search faster than O(log n)?

- Not using comparison-based searching?
- Is there anything else?
  - Hash Tables (used in Python's dictionary)

Hashing

The Goal

- *Key Idea:* Store values in an array-based structure and *map each key* (e.g. Product Code) *to a unique position* in the array called the *Hash Value.*
- *Retrieving the key is O(1)* as long as the mapping function is O(1)
- Challenge: Create a mapping so that each key gets its own spot?

Product Code key		**Hash Value** position
x10001	→	0
x10002	→	1
x10003	→	2
x10004	→	3
x10005	→	4
x10006	→	5
x10007	→	6
x10008	→	7

Hashing

A Simple Example

- Here it is easy, convert the last character of the product code to an integer, i.e. `int(pc[-1]))-1`

- A *hash function* maps the *key* to a *position* in the array.

Product Code key		**Hash Value** position
x10001	→	0
x10002	→	1
x10003	→	2
x10004	→	3
x10005	→	4
x10006	→	5
x10007	→	6
x10008	→	7

Hashing

Components

Hash Table

- An array-based structure
- O(1) random access by position
- we will call it HT[ ]

Hash Function

- Maps a *key* to a specific *position* in the *Hash Table*
- we will call it h(), i.e. h(key) = position in HT.

Collisions

- Challenge: Multiple keys may map to the same position

Hashing

Creating a Hash Table

```
HT = [None] * M  # Hash Table for M items
h(key) = key % M # use modular arithmetic
```

Example

Let M = 13 and add the value *431* to the Hash Table

- first observe that *431* = 33 x 13 + *2*
- i.e. *431* mod 13 = *2* (in Python *431* % 13 = *2*)
- so h(*431*) = *2*, i.e. our hashing function maps *431* to *2*
- so store item *431* at location *2*, i.e. HT[*2*] = *431*,

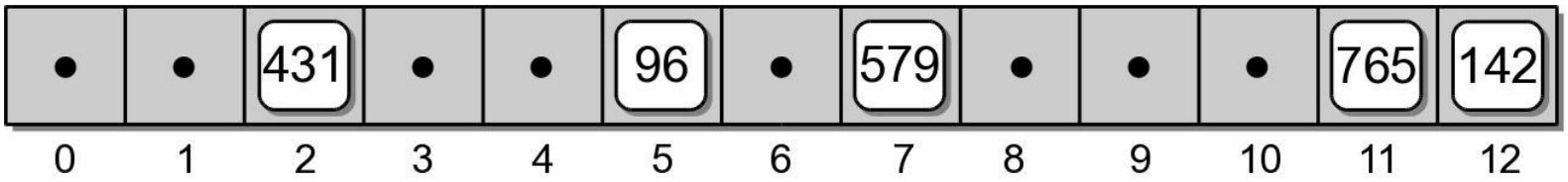
Hashing

Creating a Hash Table

Now add the values 96, 579, 765, 142 to the Hash Table

h(96) = 96 mod 13 = 5	→ store 96 at 5	HT[5] = 96
h(579) = 579 mod 13 = 7	→ store 579 at 7	HT[7] = 579
h(765) = 765 mod 13 = 11	→ store 765 at 11	HT[11] = 765
h(142) = 142 mod 13 = 12	→ store 142 at 12	HT[12] = 142

HT:

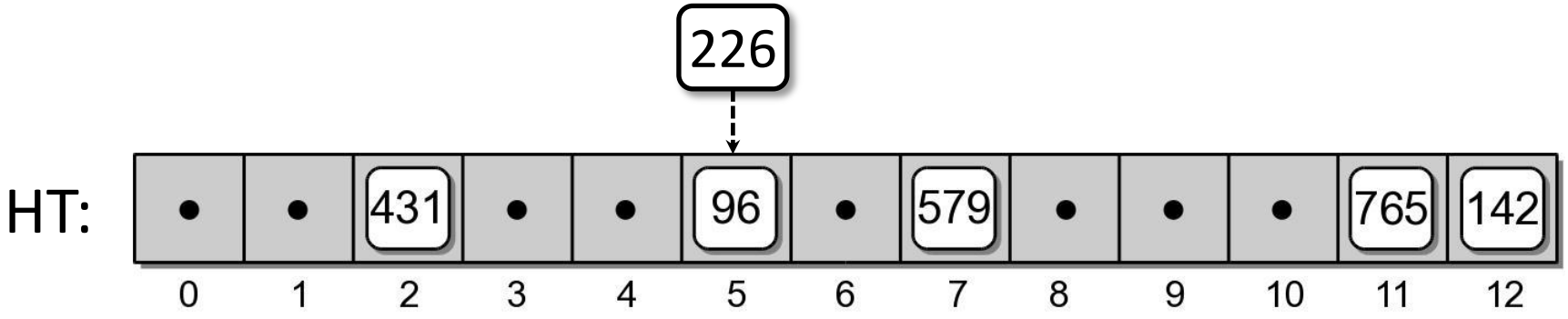


Hashing

Dealing with Collisions

Collision Example: add 226 into the Hash Table

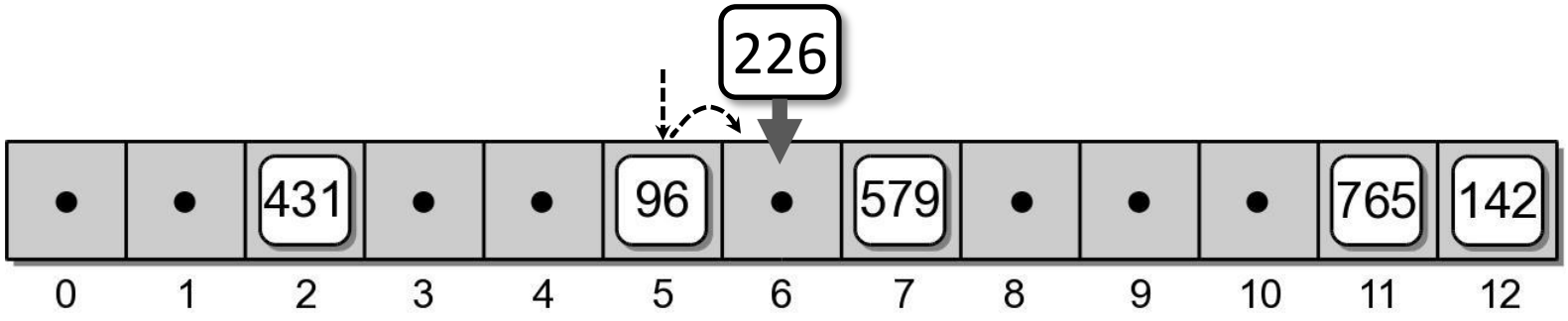
- h(226) = 226 mod 13 = 5, so store 226 at location 5
- but there is *already a value being stored at location 5,* i.e. the value 96.
- This is called a *collision*.



Resolving Collisions: Linear Probing

Linear Probing: Inserting key

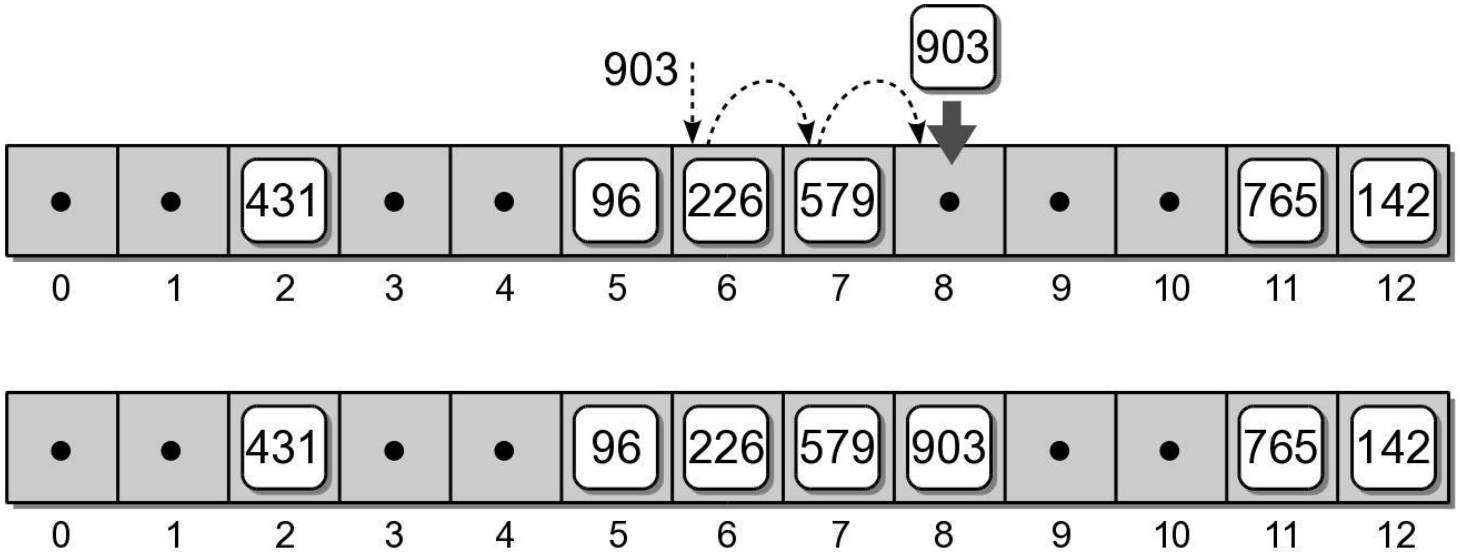
- If there is already a value at p = HT[h(key)], then try p+1, p+2, p+3, ... until there is an unused spot *Wrap around* to beginning of array, if necessary



note: Fig 11.5 (pg 312) in the course text has wrong values.

Resolving Collisions: Linear Probing

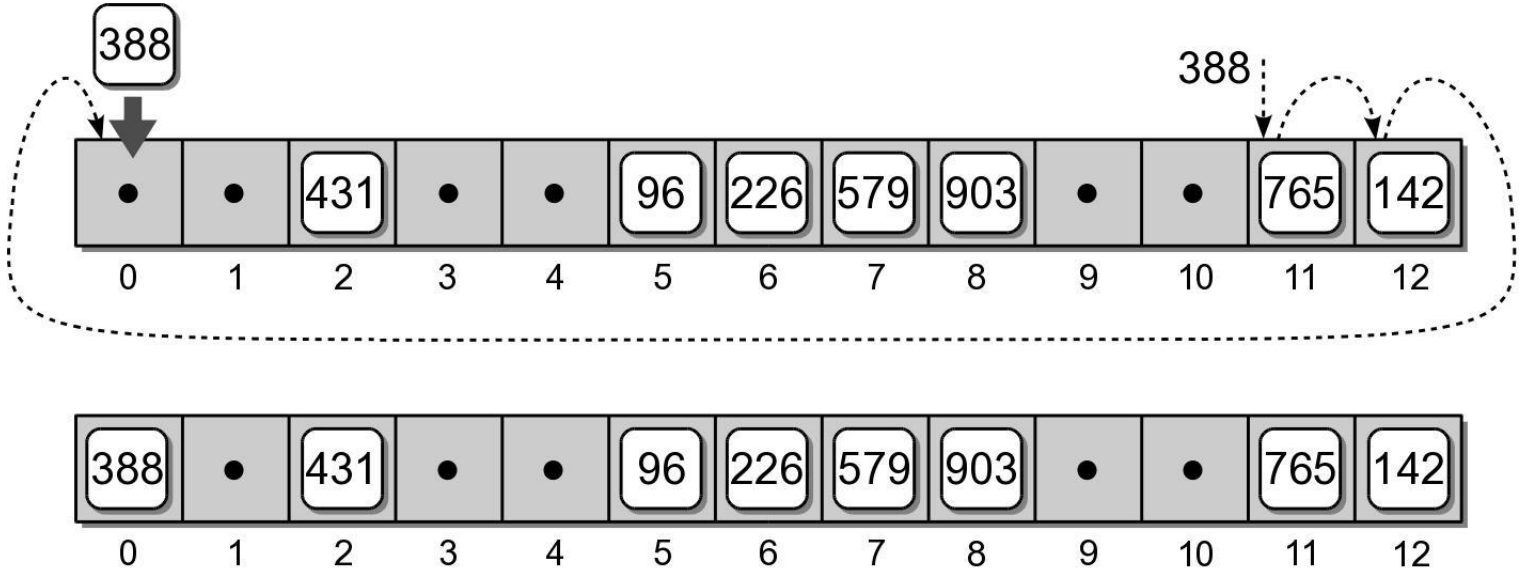
- Now add 903, first calculate h(903) = 6
- *but* 903 *collides* with HT[6] (which contains 226)
- *so* move to next position, i.e. 7
- *but* 903 also *collides* with HT[7] (which contains 579)
- *so* move to next position, i.e. 8, set HT[8] = 903



Resolving Collisions: Linear Probing

Wrap Around

- Now add 388, first calculate h(388) = 11
- *but* 388 *collides* with HT[11] (which contains 765) and HT[12] (which contains 142), so *wrap around* to position 0, and set HT[0] = 388



Hashing

Searching for a target key

- First check HT[h(*target*)]?

  - if it is the *target* → done, *target* found

  - if it is *None* → done, *target* is not in HT

  Is a different value at that location?

  - then begin a linear search

  - repeat until (*None* is found) or (*target* is found) or (you are back where you started, i.e. table is full):

    check next location

Hashing

Deleting a Key

Approach 1

- Set HT[h(key)] to *None*
- What could go wrong?
  - if key is in the middle of a cluster, it will make it look like the cluster ends at h(key)

Approach 2

- How can we fix this problem?
  - use another value to represent deleted items
  - set HT[h(key)] to ‘x’, i.e. a non-numerical value

Hashing

Expanding the Hash Table

What if HT (the Hash Table) becomes *full*?

- Create a larger array (at least twice the size)
- Add all keys to the new array
- Need to *rehash* (i.e. find new locations) as the hash function must be updated
  - Old hash function had a range of 0-12.
  - *New hash function must have bigger range.*

Hashing

Expanding the Hash Table

- Don't wait for table to become full
- α = load factor = number of keys / size of table
- Best to rehash while $\alpha < 80\%$ full
- Also, best for table size to be a prime number
  - Double, add 1, and look for next prime number
  - e.g. old 13, start looking at 2 x 13 + 1 = 27, then 29
  - Though, sometimes an odd number is sufficient

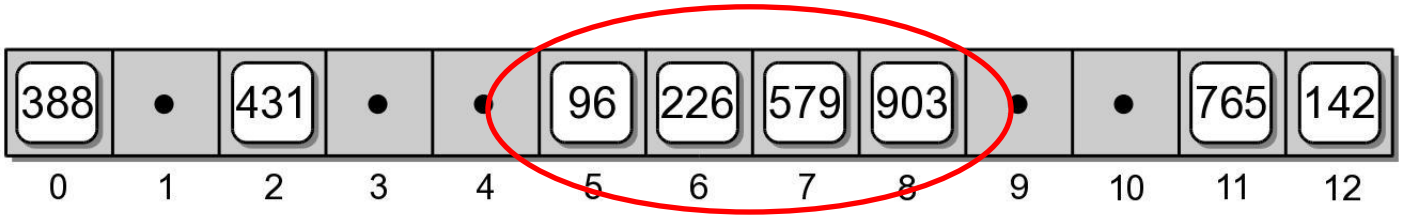
note: Fig 11.14 (pg 319) in the course text has wrong values.

Clustering

Collisions

- Clustering around a hash position is called *primary clustering*
- Have to do a linear search to find an item in a cluster.
- Five times more likely to end up in position 9 (at the end of the cluster) than at positions 4 or 10.

Cluster of size 4



Clustering

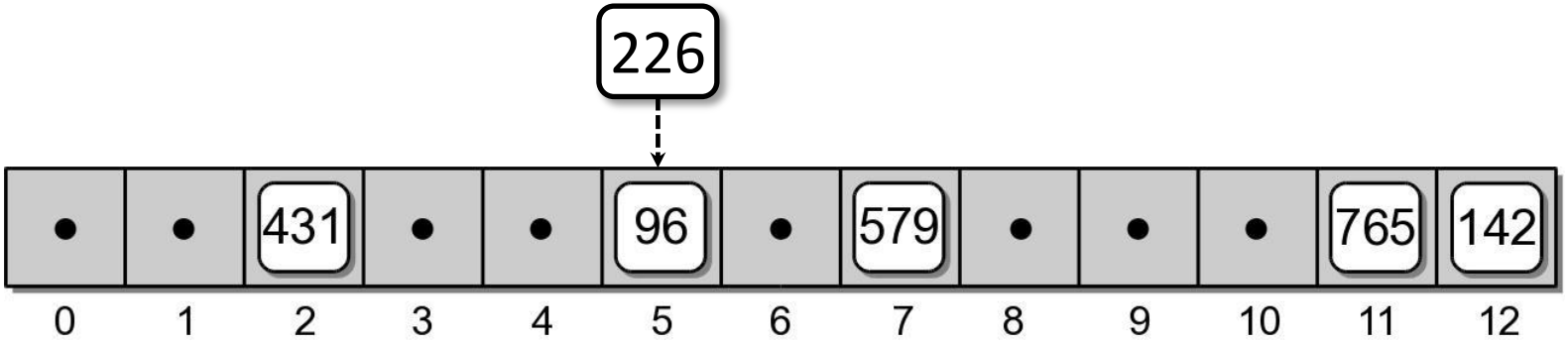
Resolving Collisions using Sequences

- Sequence *starts* at $h_k = h(key)$
  and takes *steps* $s_1, s_2, s_3, \ldots$
- Sequence h_k, $h_k + s_1$, $h_k + s_2$, $h_k + s_3$, … should
  - reduce primary clustering
  - ideally cover the entire table
  - if the table size is prime, any step size works
  - if table size was 16 then a step size of 8 would only step between two positions, this is a bad choice

Clustering

Modified Linear Probing

- Instead of stepping by 1, try a larger step c
- Leaves spaces between search positions
- If M is prime, $\boldsymbol{h}_k$, $\boldsymbol{h}_k$+c, $\boldsymbol{h}_k$+2c, $\boldsymbol{h}_k$+3c, … (all mod M) covers the entire table
- E.g. we have the values 431, 96, 579, 765, 142 in the Hash Table and we want to add 226

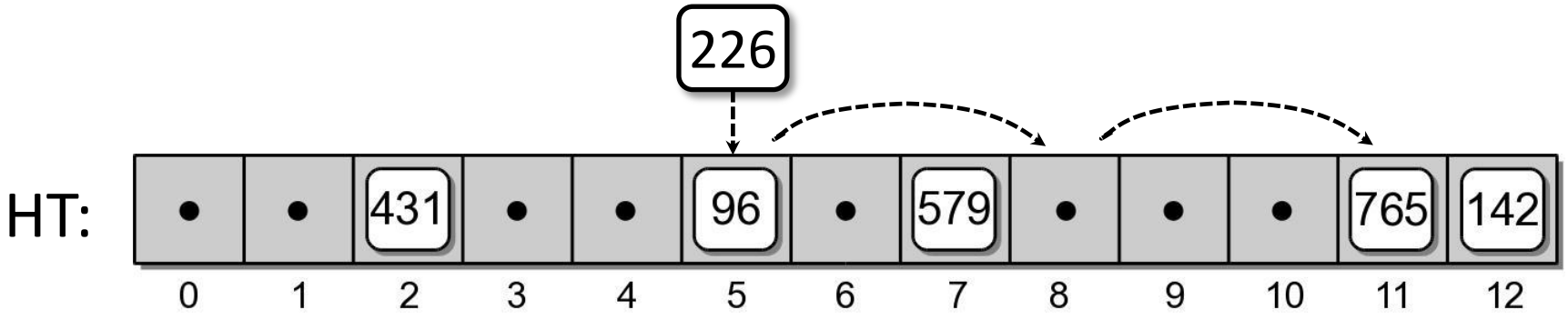


Clustering

Modified Linear Probing

Add 226 into the Hash Table

- h(226) = 5 which collides with the value 96.
- If the step size *c* = 3, then we would consider the locations 5, 8, 11, 1, 4, etc. (note that 14 mod 13 = 1)



Clustering

Quadratic Probing

- Instead of stepping by a constant each time,
- try increasing larger steps sizes from the original hash position , i.e. 1, 2^2, 3^2, 4^2, …
- e.g. $\boldsymbol{h}_k$, $\boldsymbol{h}_k+1$, $\boldsymbol{h}_k+2^2$, $\boldsymbol{h}_k+3^2$, $\boldsymbol{h}_k+4^2$, …
- Can result in secondary clustering
  - Two keys map to the same index, and their probe sequence is identical
- Only likely to reach half of the positions, but is often effective

Clustering

Double Hashing

- Suppose we have a collision at $\boldsymbol{h_k}$
- *Key Idea:* Choose a step size which *depends on the key* itself (i.e. differs with different keys)
- A second hash function provides the step size, ***hp***

  step size = $\boldsymbol{hp(key) = 1 + (key \% P)}$, for some P < M
- try $\boldsymbol{h_k = i * hp(key)}$ for i = 0, 1, 2, 3, ... until an available position is found
- As always, calculate positions mod M

Clustering

Double Hashing

- steps are $h_k = i * hp(key)$ for $i = 0, 1, 2, 3, \ldots$
- If M is prime, then all values will be reached by such a probe sequence

Example

- M = 13
- h(*key*) = *key* mod 13
- hp(*key*) = 1 + (*key* mod 8)

  Note: we always add 1 when calculating ***hp*** because we don't want a step size of 0.

Clustering

Double Hashing

Have values 431, 96, 579, 765, and 142 in the Hash Table

- try to add 226,
- h(226) = 5, *collision* with 96
- try $\boldsymbol{h_k} = \boldsymbol{i} * \boldsymbol{hp(key)}$ for $i = 1$
- hp(226) = 1 + (226 mod 8) = 1 + 2 = 3, *so step size is 3*
- try h(226) + (1 x 3) = 5 + 3 = 8

0	1	2	3	4	5	6	7	8	9	10	11	12
•	•	431	•	•	96	•	579	226	•	•	765	142

Clustering

Double Hashing

Have values 431, 96, 579, 765, 142, 226 in the Hash Table

- try to add 338,
- h(388) = 11, *collision* with 765
- try $\boldsymbol{h_k = i * hp(key)}$ for $i = 1$
- hp(388) = 1 + (388 mod 8) = 1 + 4 = 5, *so step size is 5*
- h(388) + (1 x 5) = 11 + 5 ≡ 3 mod 13, store in position 3

0	1	2	3	4	5	6	7	8	9	10	11	12
•	•	431	•	•	96	•	579	226	•	•	765	142

Hashing

Efficiency

- Depends on
  - Size of hash table
  - Hash function
  - Type of probe to resolve collisions
  - Load factor α (how full it is)
- Operations: insert / delete / retrieve
- Note: *search is primary to all operations*

Hashing

Efficiency

- $\alpha = n/M$ (i.e. # item / size of table)
- Best Case: Search is O(1)
  Each key maps to a unique position
- Worst Case: Search is O(M)
  - Each key maps to the same position or a conflict
  - All locations must be checked
- Can derive formulas for average number of comparisons for successful and unsuccessful searches (though we won't)

Hashing

Efficiency: Average Case

- *Assume:* keys uniformly distributed
- *Calculate*: average search times

Load Factor α	0.25	0.5	0.67	0.8	0.99
Successful search:					
Linear probe	1.17	1.50	2.02	3.00	50.50
Quadratic probe	1.16	1.33	1.65	2.01	6.71
Unsuccessful search:					
Linear probe	1.39	2.50	5.09	13.00	5000.50
Quadratic probe	1.33	2.00	3.03	5.00	100.00

note: Table 11.1 (pg 321) in the course text has wrong values.

Separate Chaining

A different approach to Collisions

- Hash table build as before, except each position *references a singly linked list (SLL)*
- When inserting/deleting/retrieving : use a hash function to find the associated linked list (bucket)
- *Inserting*: add to front of SLL
- *Retrieving*: use linear search through SLL
- Deleting: remove from SLL after linear search
- Called *separate chaining.*

Separate Chaining

Terminology

- Also called *open hashing*

  Values stored outside the table

- Also called *closed addressing*

  Values are stored at the hash index

Separate Chaining

Efficiency

- Assuming numbers are *uniformly distributed*
- Average SLL length is $\alpha = n/M$
- Average comparisons:

  Successful searches: $1 + \alpha/2$

  Unsuccessful searches: $1 + \alpha$
- If $\alpha < 2$, average case is 1 comparison
- What is a disadvantage of this approach?

  - need extra memory

Good Hash Functions

Important Properties

- Easy and fast to calculate
- Not random (reproducible)
- Depends on value of key
- Table size should be a prime number

Good Hash Functions

Common approaches

- *Division mod M* (i.e. % M in Python)
- *Truncation*: choose certain digits/characters and combine , i.e. pick 1st and 3rd digit, h(4567) = 46
- *Folding:* break key into pieces and combine

  i.e. add 1st and 2nd half, h(4567) = 45 + 67 = 112
- *Strings*

  sum ASCII/Unicode values and treat as integer

  combine individual ASCII codes in a polynomial

Another approach for Searching

Introducing Search Trees

- Organize data is a nonlinear manner
- Place restrictions on storage format

  → *Search trees*
- Advantage: May speed up search
- Disadvantage: Organization may require more memory and overhead
- Recall: as before, we will just discuss keys, rather than additional, related data stored in the tree

Topic 13 – Binary Search Trees

Key Concepts

- Section 14.1: Searching, Inserting, Deleting

Recall

- Topic 10: talked a bit about Binary Trees

Mini Review

Sorted Python List (a.k.a. array): *great for* binary search, O(log*n*), but *poor for* insert and delete, O(n) .

Linked List: *great for* insert and delete, O(1), but poor for search, O(n).

Hash Table: *great for* average case search, amortized insert and delete O(1) but *poor for* worst case O(n).

Introducing ... *Binary Search Trees (BST)*

Binary Search Trees

Structure (a.k.a. BST invariant, BST Property)

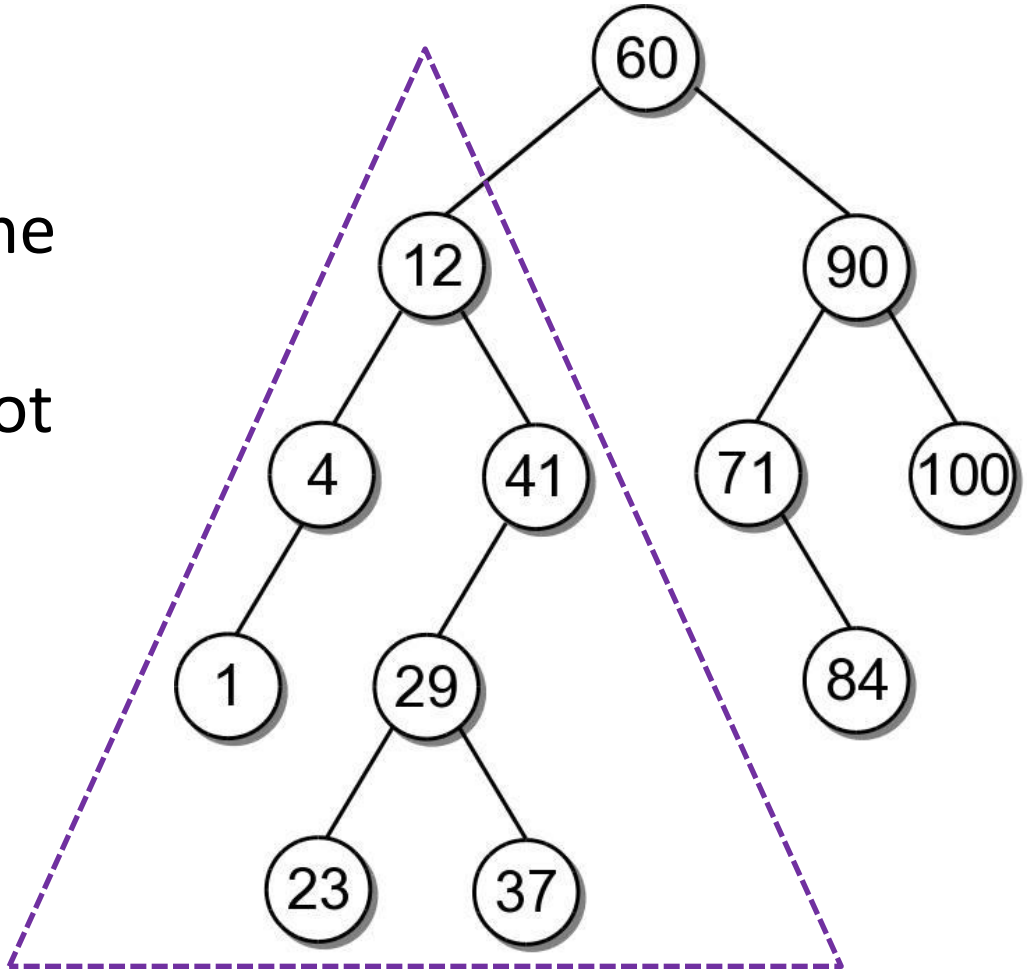
The structure of a binary search tree (BST) is either

- empty or
- a *key* value, a *left subtree* and a *right subtree*, where
  - all keys in the left subtree are less than the root key
  - all keys in the *right subtree* are greater than the root key
  - the left and right subtrees are both BST
- *Assume*: no duplicate keys → makes code simplier.

Binary Search Trees

Structure

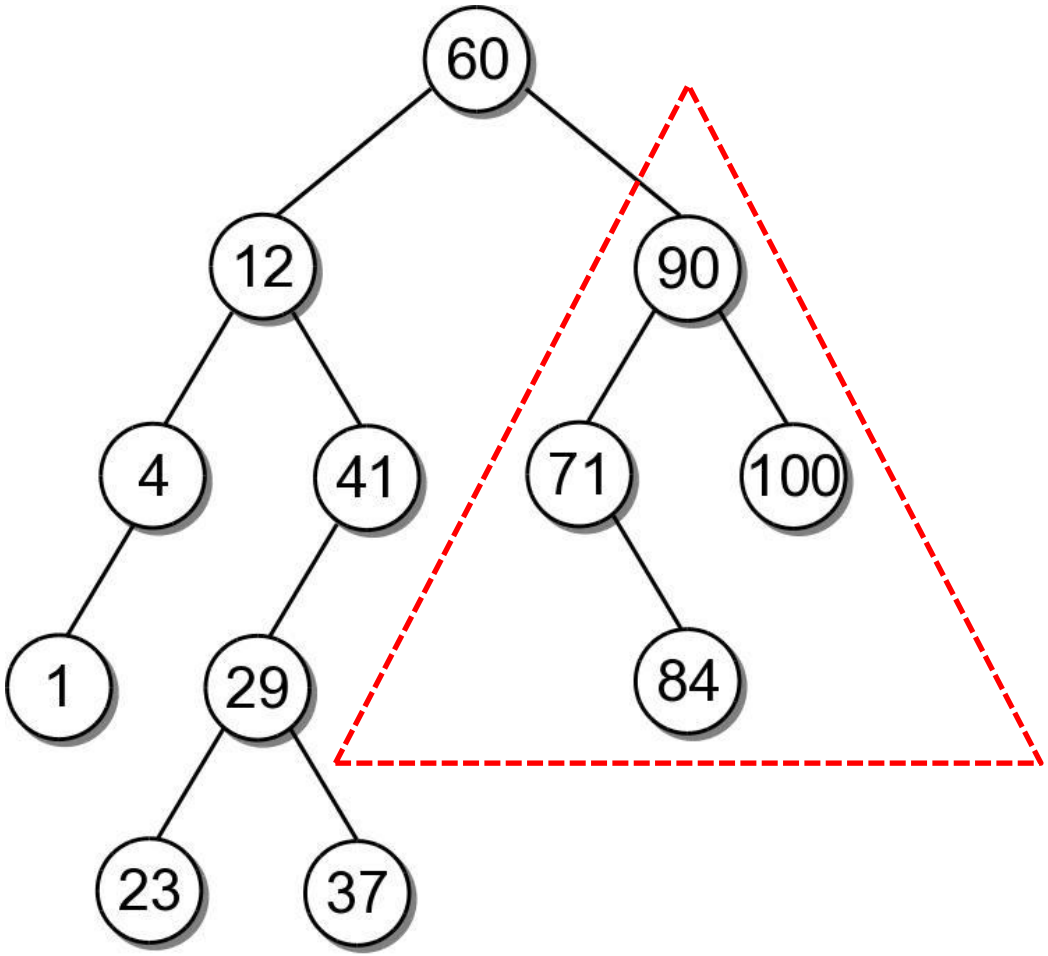
All the keys in the *left subtree* are *less than* the root key, 60.



Binary Search Trees

Structure

All the keys in the *right subtree* are *greater than* the root key, 60.



Binary Search Trees

Structure

- Similar to a doubly linked list → have two references
- for BST called *left* and *right* (not *next* and *prev*)

```
class BSTNode:
    def __init__(self, key, value)
        self.key = key
        self.value = value
        self.left = None
        self.right = None
```

Simplification: to keep the diagrams simple we will *ignore self.value* for now.

Binary Search Trees

Structure

- Where is the *minimum value* → the *leftmost* node

```
n = self.root
while n.left != None:
    n = n.left
return n.key
```

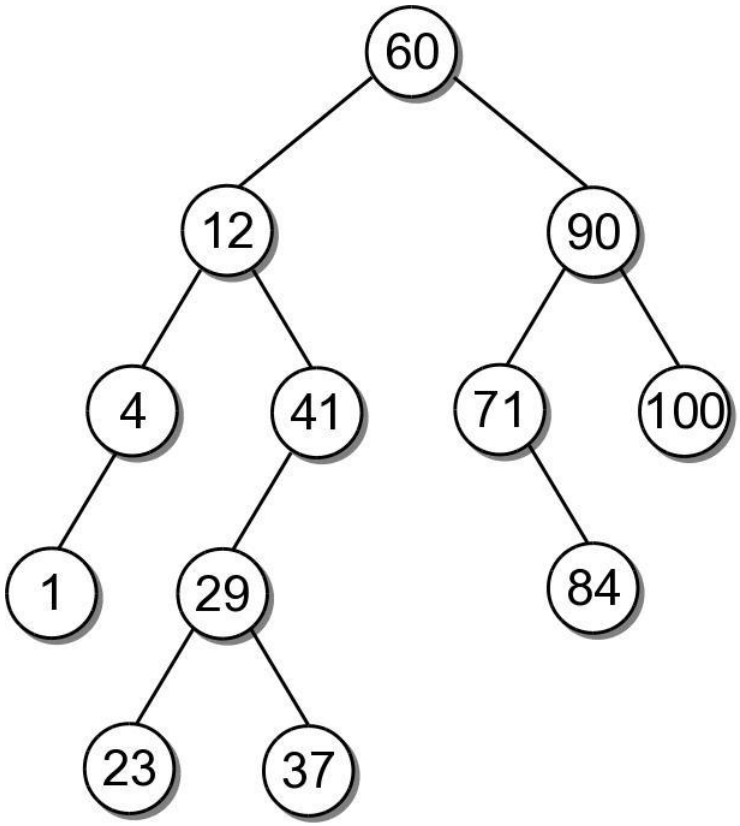
- Where is the *maximum value* → the *rightmost* node

```
n = self.root
while n.right != None:
    n = n.right
return n.key
```

Binary Search Trees

In-order Depth First Search

- recall from slides 207-209
- have three ways of traversing trees depth first
- *the names describe the order you process parent vs. children*
  - pre-order: parent first
  - post-order: parent last

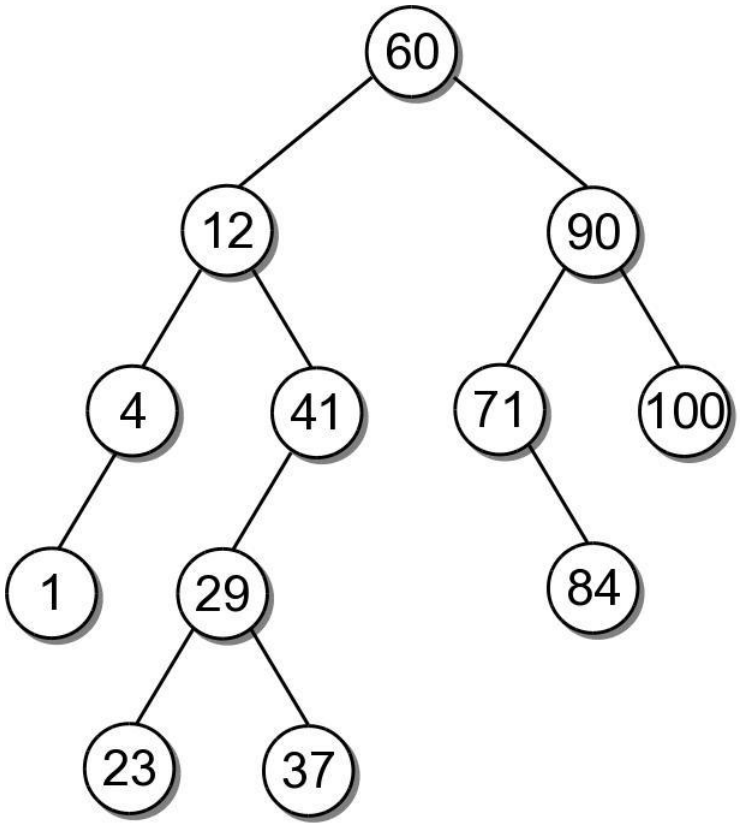


Binary Search Trees

In-order Depth First Search

- in order = process left child, parent, right child

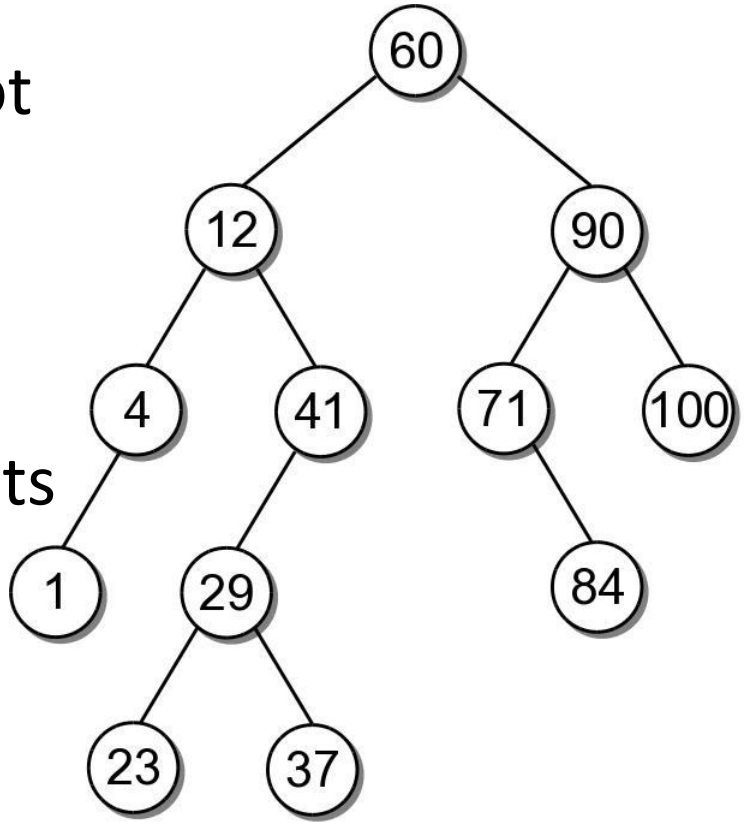
```
def inorder(subtree):
  if subtree != None:
    inorder (subtree.left)
    print subtree.key
    inorder(subtree.right)
```



Binary Search Trees

In-order Depth First Search

- but BST always has
  - keys with values *less than* root in *left subtree*,
  - keys with values *more than* root in the *right subtree*
- in-order depth first search prints out nodes in ascending order
- 1, 4, 12, 23, 29, 37, 41, 60, ...



Binary Search Trees

Searching a BST

Start at root and go left or right until either the key is found or there is no subtree (i.e. *n* is None)

```
n = root
while n != None:
    if n.key == target:
        return n.value
    elif target < n.key:
        n = n.left
    else # target > n.key
        n = n.right
return None
```

Binary Search Trees

Searching a BST - Example

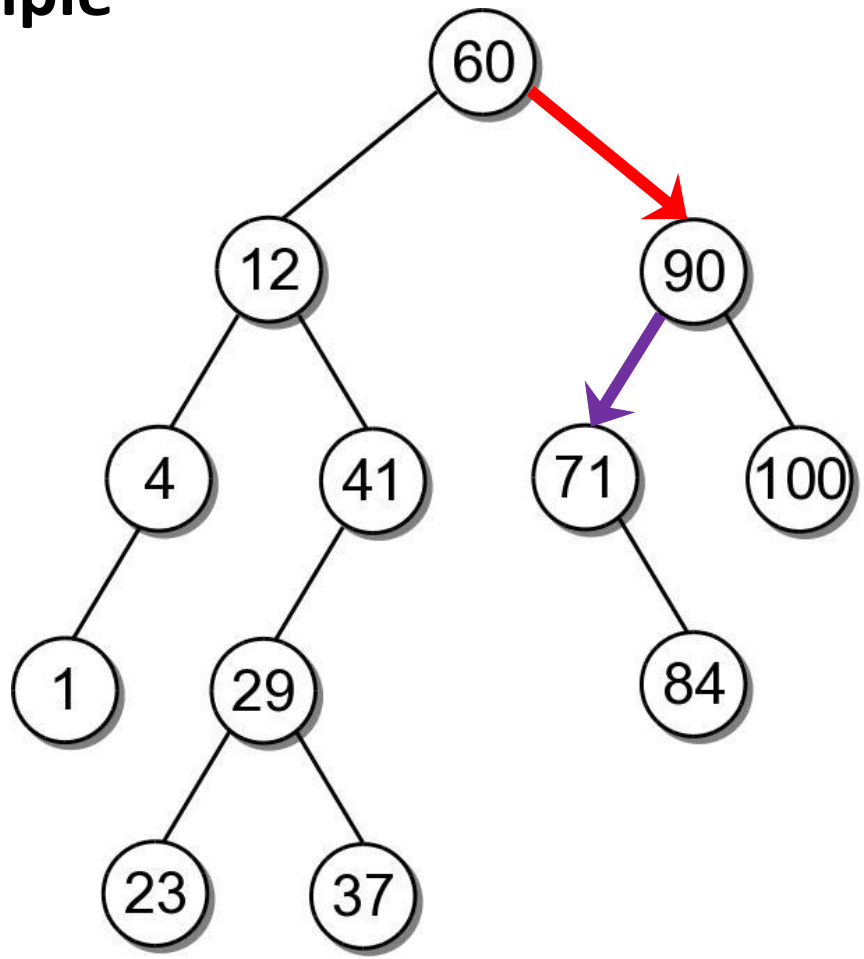
find(71)

start at root

71 > 60 → go right

71 < 90 → go left

found it



Binary Search Trees

Searching a BST - Example

find(43)

start at root

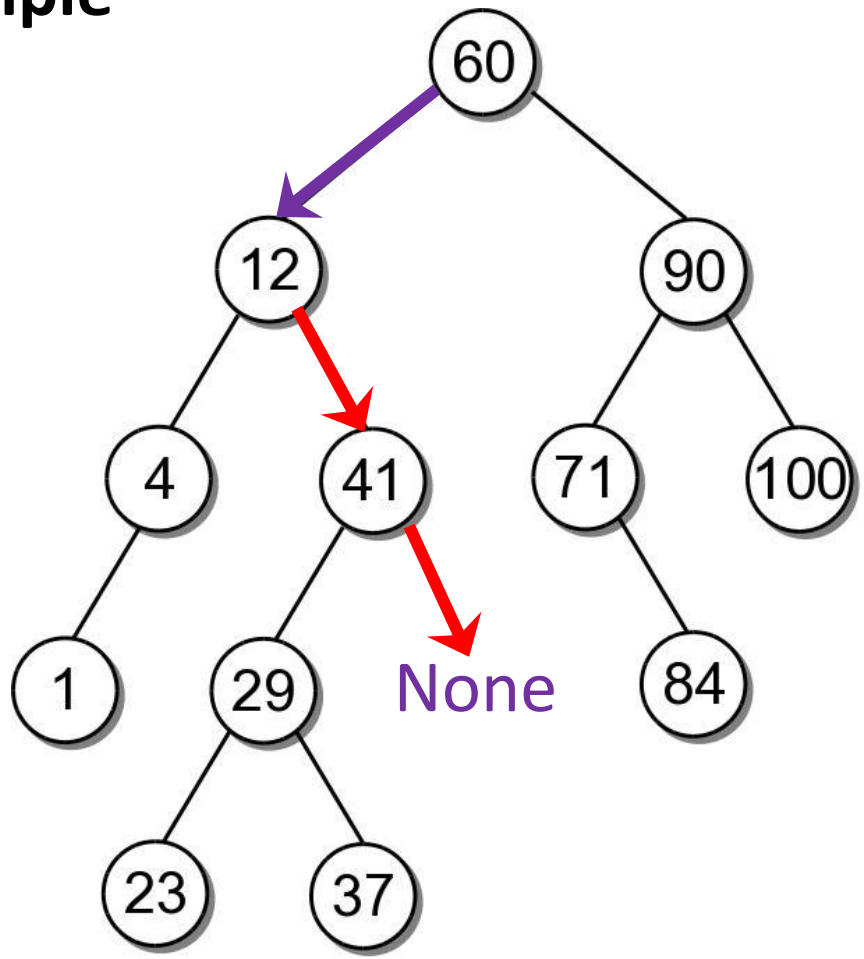
43 < 60 → go left

43 > 12 → go right

43 > 41 → go right

right is None

43 is not in BST



Binary Search Trees

Creating a BST

Key Idea: Repeat the insert method for each key

- For isert there are three steps...
  1. *Search for* the appropriate location, note where the search fails is exactly where it belongs!
  2. *Create* a node for the new key
  3. *Link* the new node into the existing BST

Constraint: must ensure that the tree maintains the *BST Property*, so... how do you insert a key?

Inserting into a BST

Basic Structure of Inserting into a BST

Three cases:

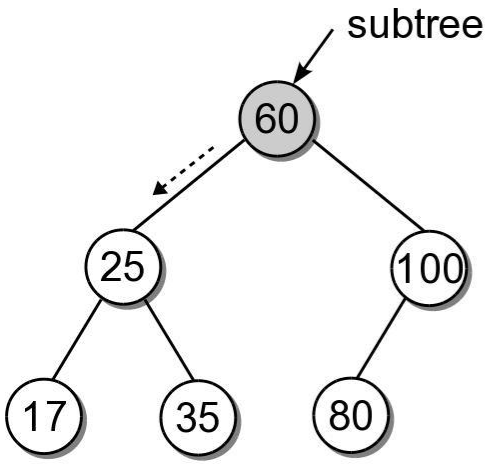
- if the tree is *empty*,
  create leaf node with key as root
- if *new key* < key at *root*,
  insert into left subtree
- if *new key* > key at *root*,
  insert into right subtree

Inserting into a BST

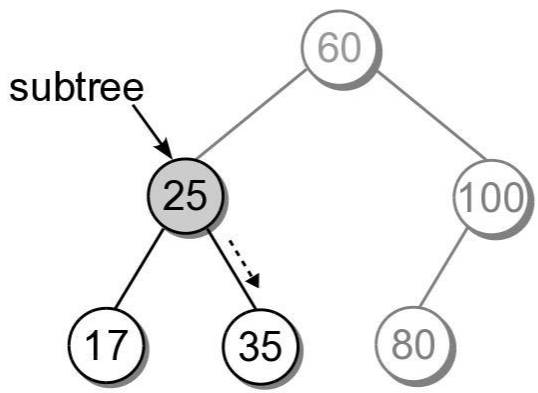
Example: add 30 to an existing tree

1. *Search for* the appropriate location (by searching the BST for 30)

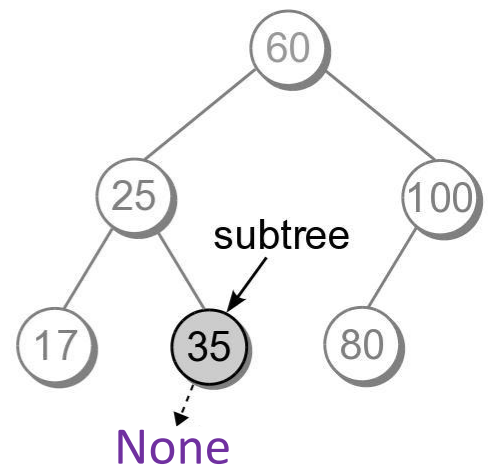
30 < 60
go left



30 > 25
go right



30 < 35
go left

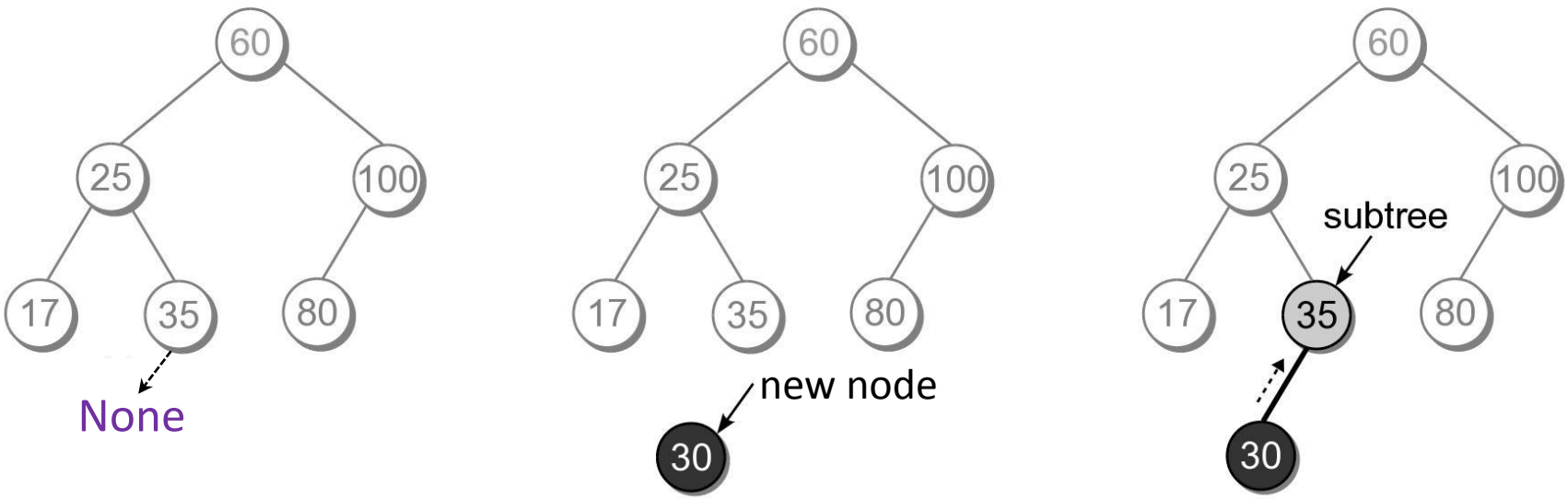


Inserting into a BST

Example: add 30 to an existing tree

2. *Create* a ned node for the new key, 30.

3. *Link* the new node into the existing BST.



Deleting from a BST

Basic Structure of Deleting from a BST

For each key there are three steps...

1. *Search* for the target key
2. *Remove* the key
3. *Reorganize* the keys in the tree, if needed, to maintain *BST Property* (left children less than root, right children greater than root)

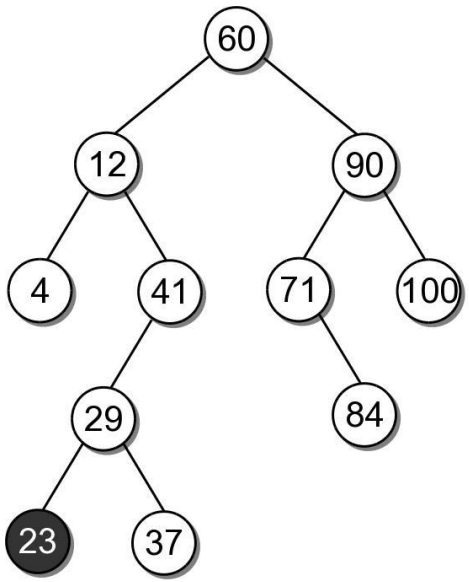
Goal: make as few changes as possible

Deleting from a BST

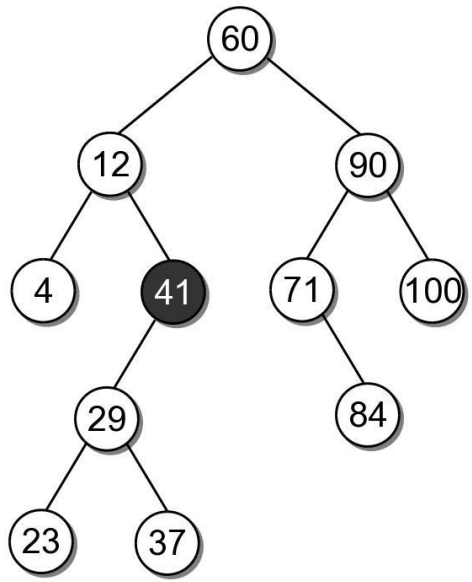
Basic Structure of Deleting from a BST

Three cases to consider...

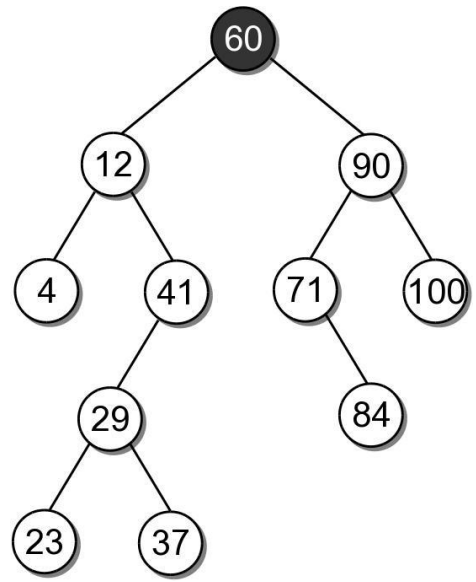
1. the node is a *leaf*



2. the node has *one child*



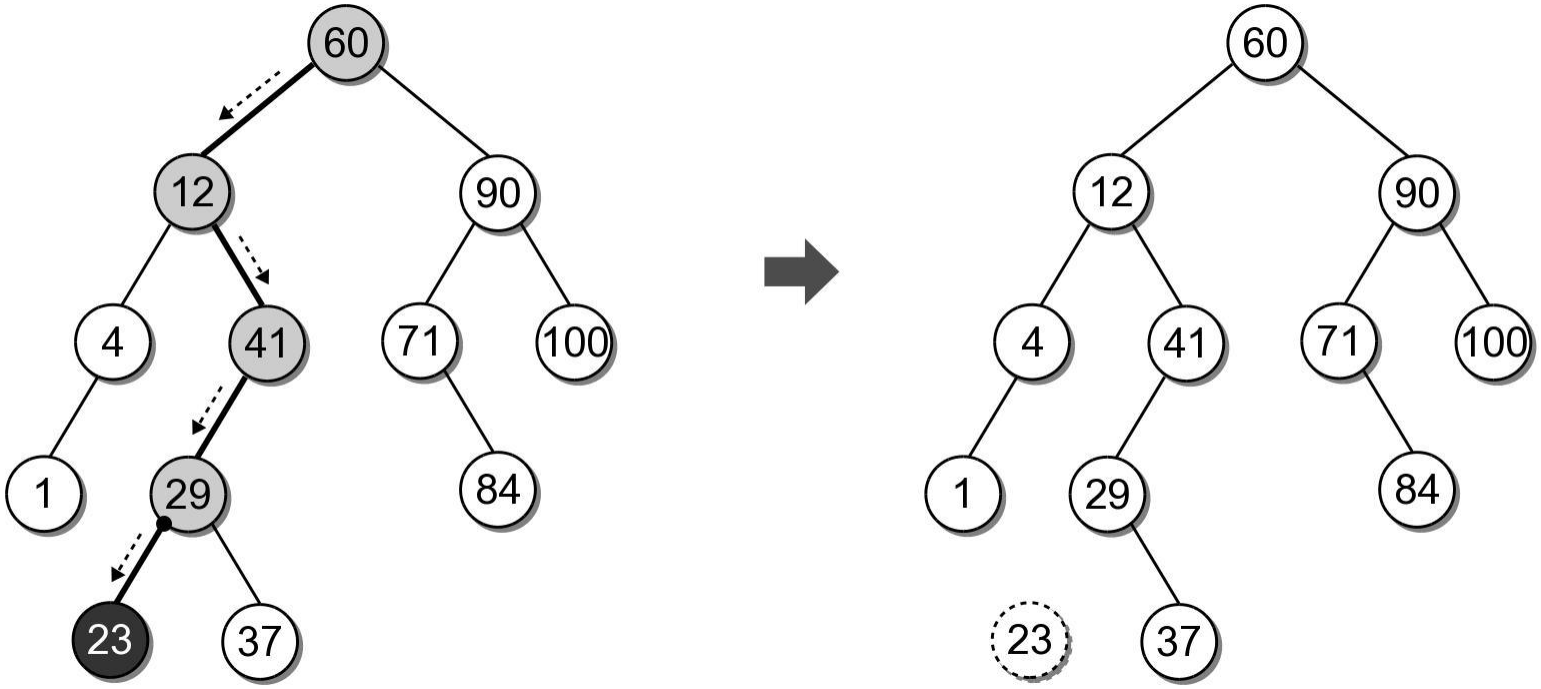
3. the node has *two children*



Deleting from a BST

Case 1: deleting a *leaf node* (e.g. 23)

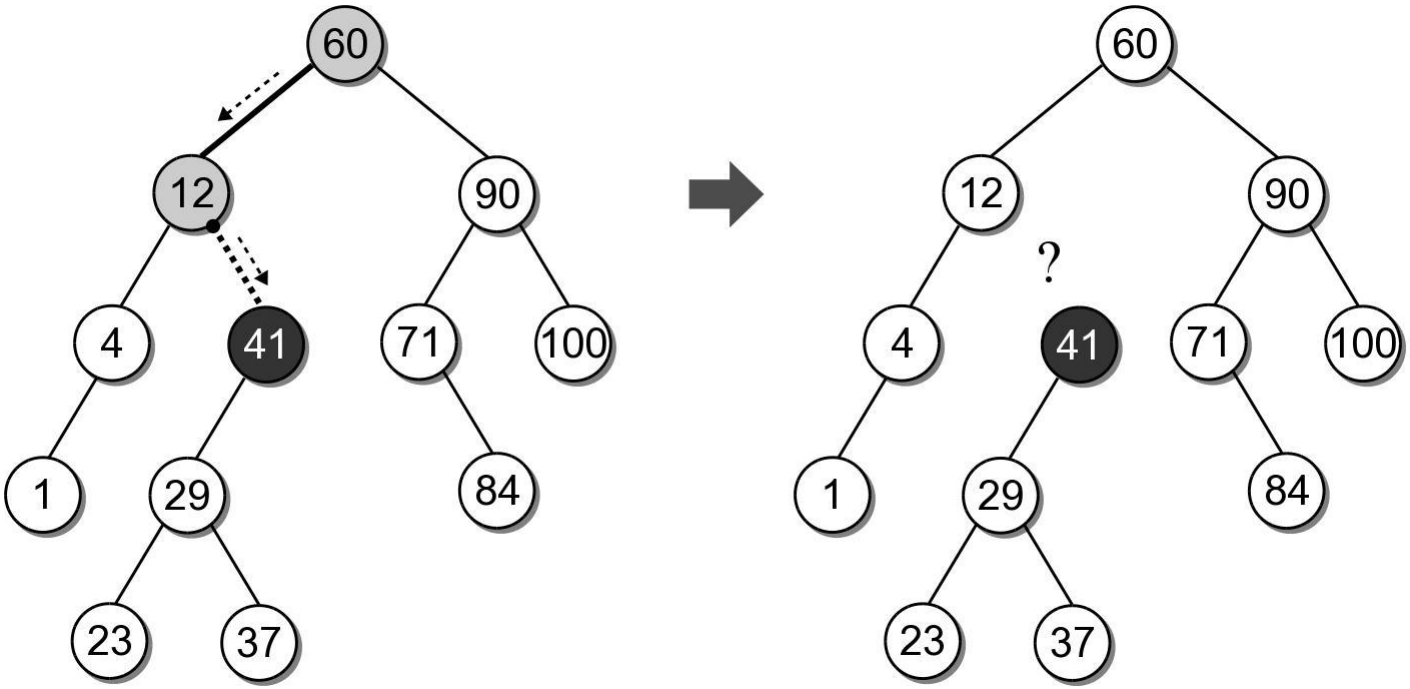
- simply *search* and *remove*
- no *reorganization* needed



Deleting from a BST

Case 2: deleting a node with a *single child* (e.g. 41)

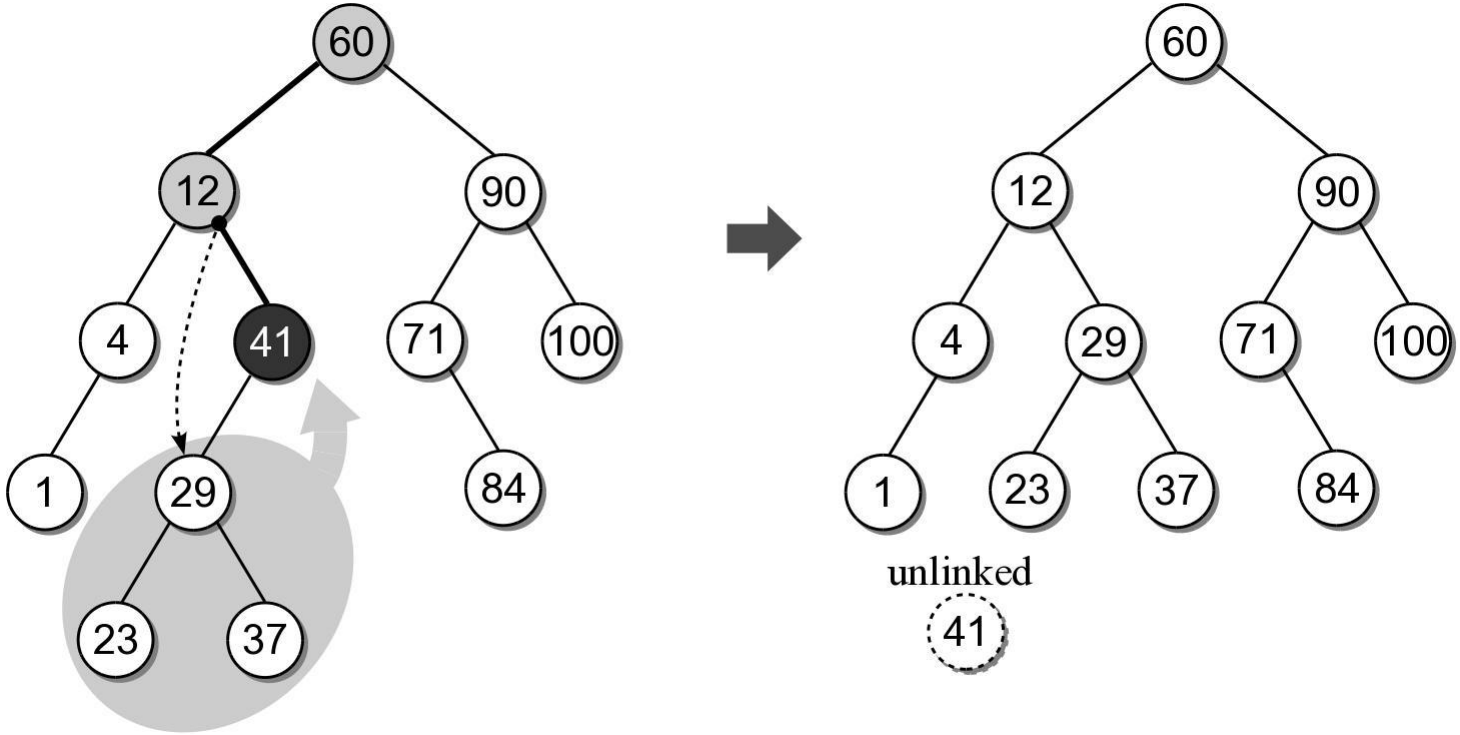
- *search* for and *remove* the node



Deleting from a BST

Case 2: deleting a node with a *single child* (e.g. 41)

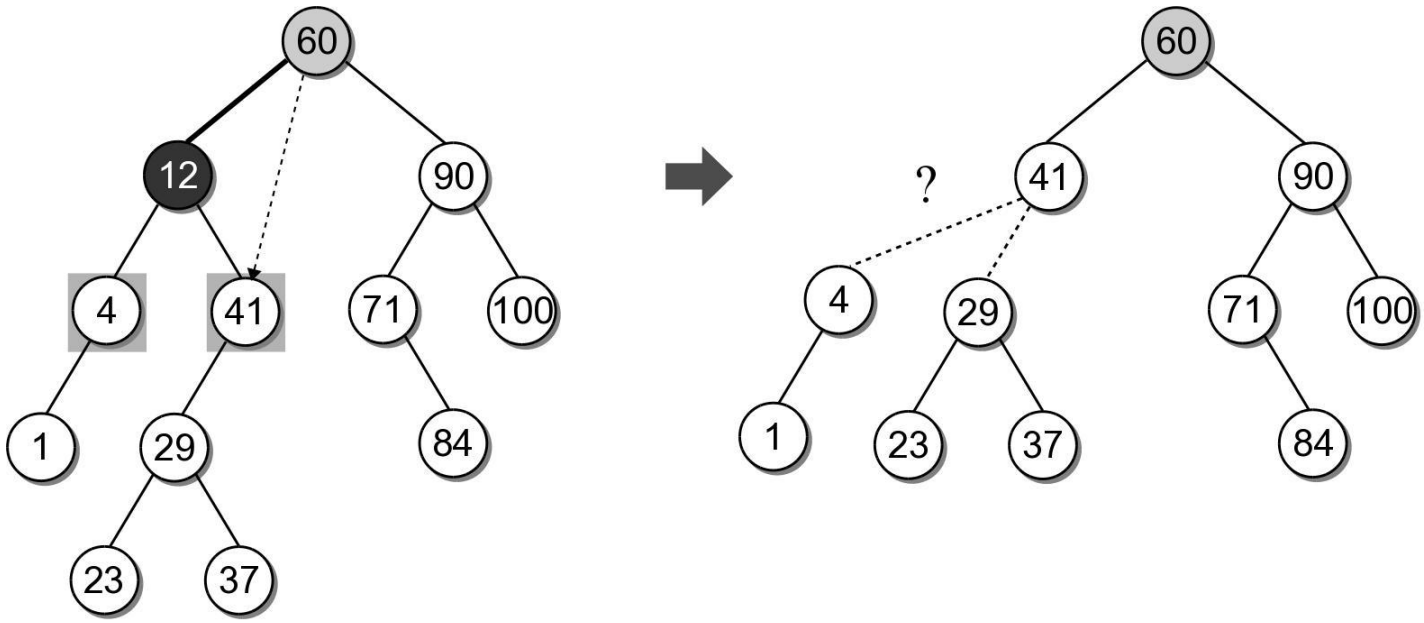
- *reorganize:* link the nodes only child to its only parent



Deleting from a BST

Case 3: deleting a node with a *two children* (e.g. 12)

- *search* for and *remove* the node, as before, *but*...



cannot replace parent (12) by child (41) → it will *violate the BST property*: 29 is a right child but is less that 41.

Deleting from a BST

Case 3: deleting a node with a *two children*

Reorganize the tree

- *Key Task*: need a new root for the subtree
- may use either:
  - *largest* value from the *left subtree*
    - rightmost value in the left subtree
    - in-order predecessor
  - *smallest* value from the *right subtree*
    - leftmost value in the right subtree
    - in-order successor

Deleting from a BST

Case 3: deleting a node with a *two children*

Steps for *reorganizing* the tree

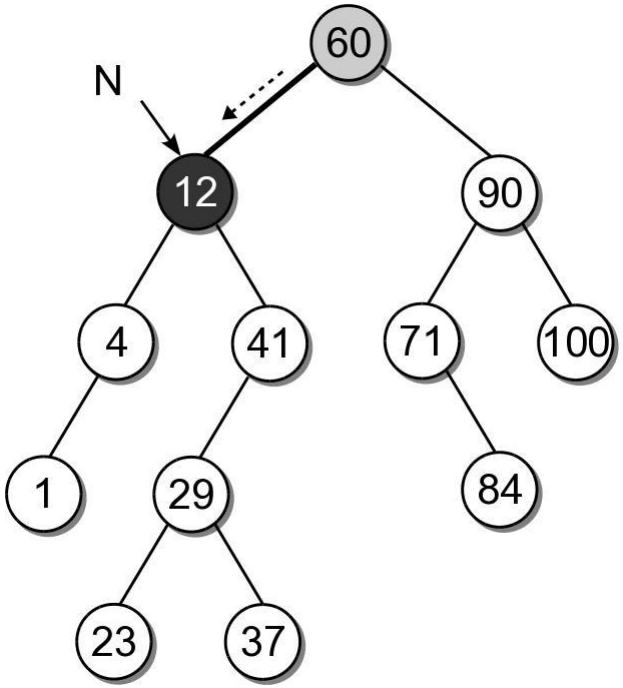
1. *Find* replacement value
2. *Set* subtree root key to the replacement value
3. *Delete* the replacement value from the left or right subtree, as appropriate

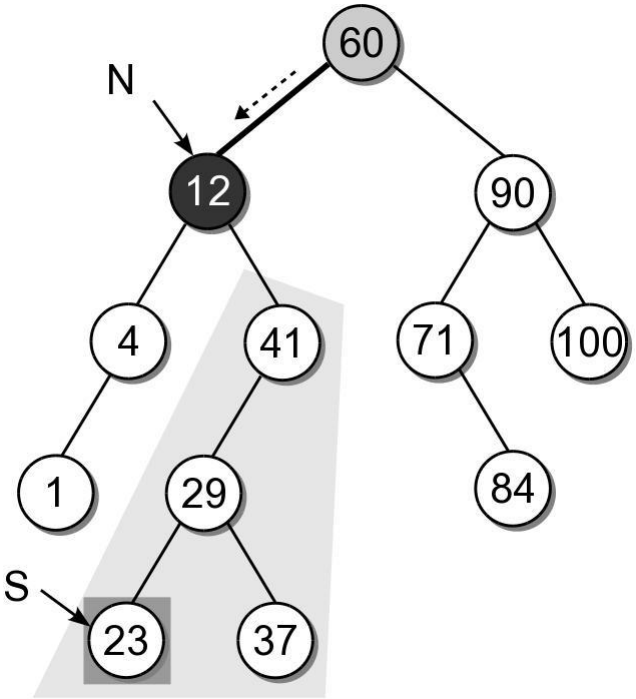
Key point: largest value in left subtree and smallest value in the right subtree *are guaranteed* to either be a leaf or only have one child.

Deleting from a BST

Case 3: deleting a node with a *two children* (e.g. 12)

1. *find replacement value:* in this case, find the smallest value in the right subtree, i.e. 23.

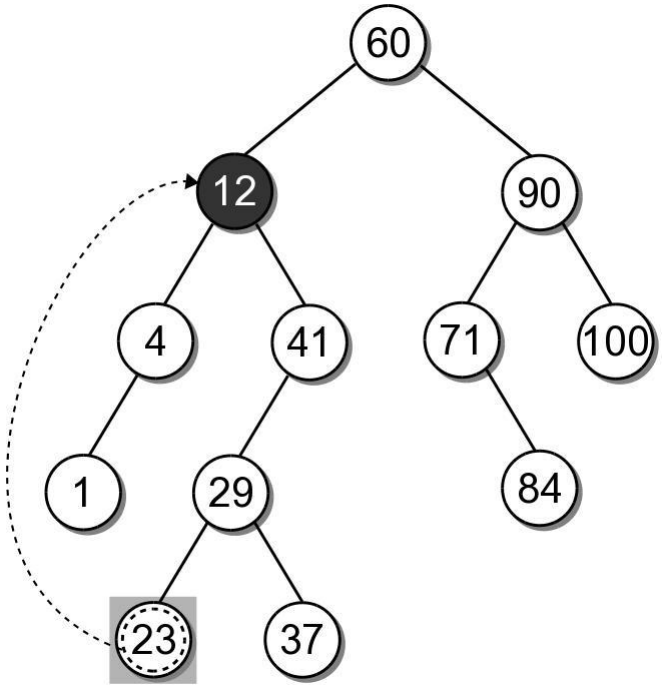


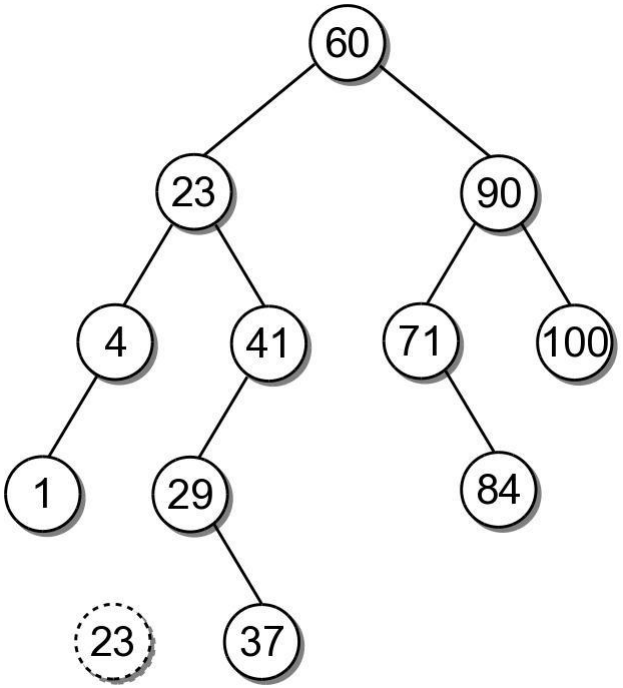


Deleting from a BST

Case 3: deleting a node with a *two children* (e.g. 12)

2. *Set* the subtree root key to the replacement value and remove the original node with the replacement value.





Binary Search Trees

Efficiency of BST Operations

Search

Worst case: make one comparison for each level of the tree, O(n)

Insert

Worst case: search entire tree and then create new node, O(n)

Delete

Worse case: search entire tree for target, then replace and update: O(n)

Binary Search Trees

Improving Performance

- The worst cases are *degenerate trees*
- *Degenerate trees* look like linked lists
- The fast cases are *balanced trees* (all the leaves are at the same level), performance would be much better

Solution: 2-3 trees

- Add an another operation to BST to maintain balance
- Improved performance
- Significantly more complicated to maintain

Topic 14: 2-3 Trees

Key Concepts

- Section 14.4: Searching, Inserting, Deleting

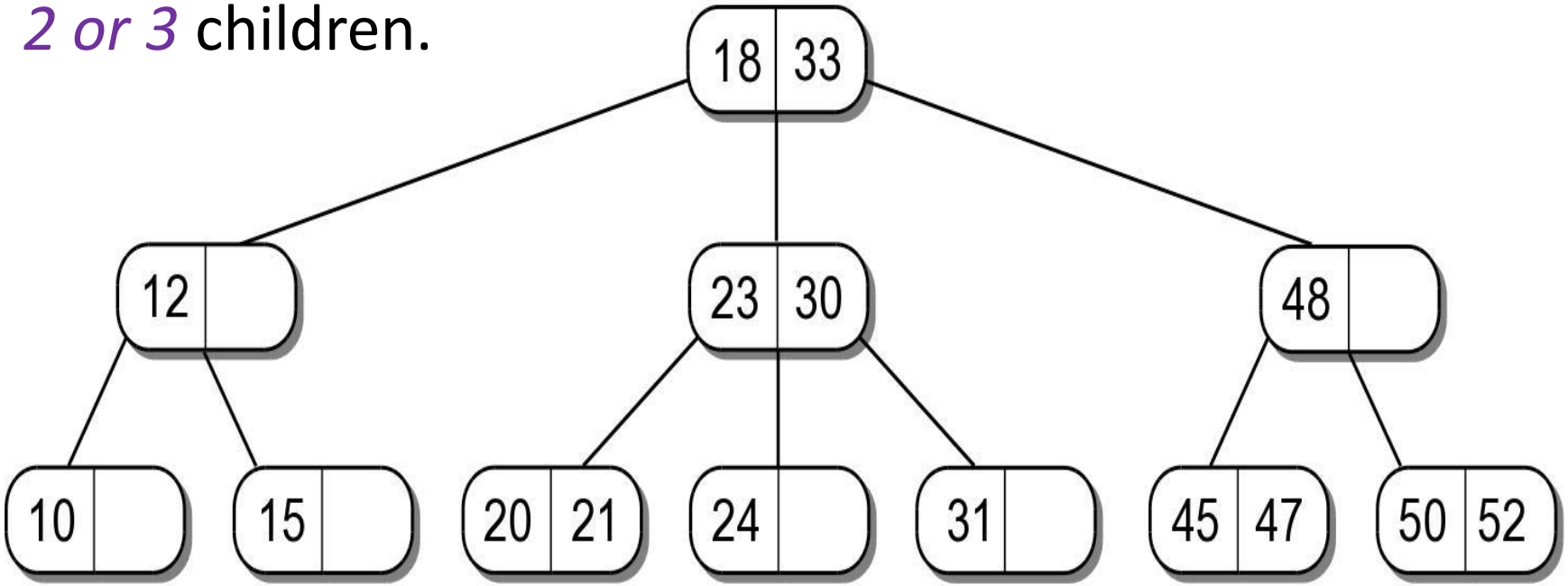
Recall

- Topic 10: talked a bit about Binary Trees
- Topic 13: talked about Binary Search Trees

2-3 Trees

Example

Nodes may have
2 or 3 children.



2-3 Tree Shape Properties

Properties

- 2-3 trees are balanced
  - all leaves are at same level
- Internal nodes may have either:
  - 1 key and 2 children or
  - 2 keys and 3 children
- Leaves can have 1-2 keys
- Recursive: Left, Middle, Right subtrees are all 2-3 trees
- Similar to Binary Search Trees but now (potentially) have *more keys* and *more children*

Review: Binary Search Trees

Structure

- recall (slide 349) the Binary Search Tree node

```
class BSTNode:
    def __init__(self, key, value)
        self.key = key

        self.value = value

        self.left = None

        self.right = None
```

2-3 Trees

Structure

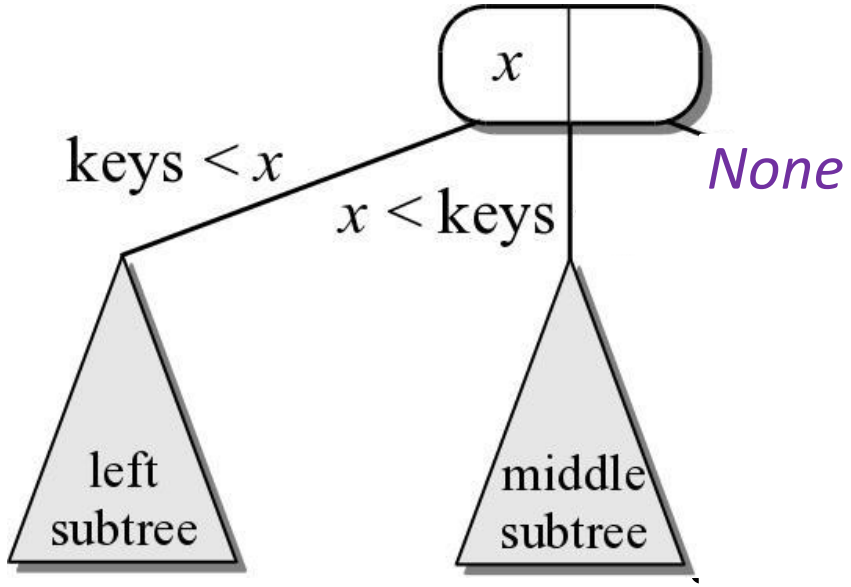
- added a 2nd key, a 2nd value and a middle child

```
class BSTNode:
    def __init__(self, key, value)
        self.key1 = key
        self.key2 = None
        self.value1 = value
        self.value2 = None
        self.left = None
        self.middle = None
        self.right = None
```

2-3 Tree Order Properties

Properties

For an internal node with 1 key and 2 children



Right subtree is empty

Just like a binary tree!

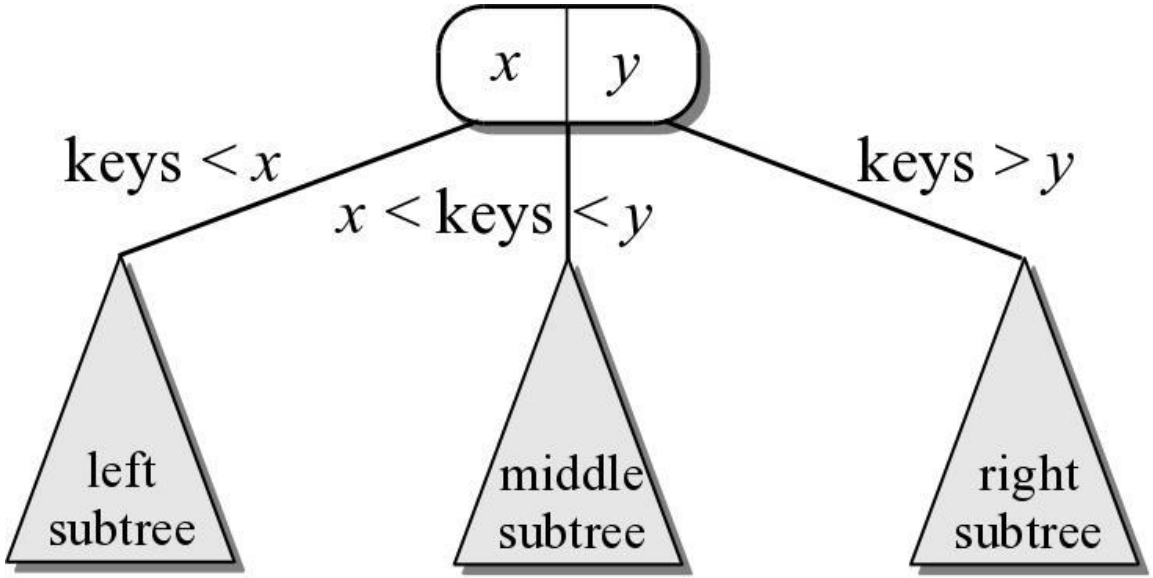
left subtree contains keys < *x*

middle subtree contains keys > *x*

2-3 Tree Order Properties

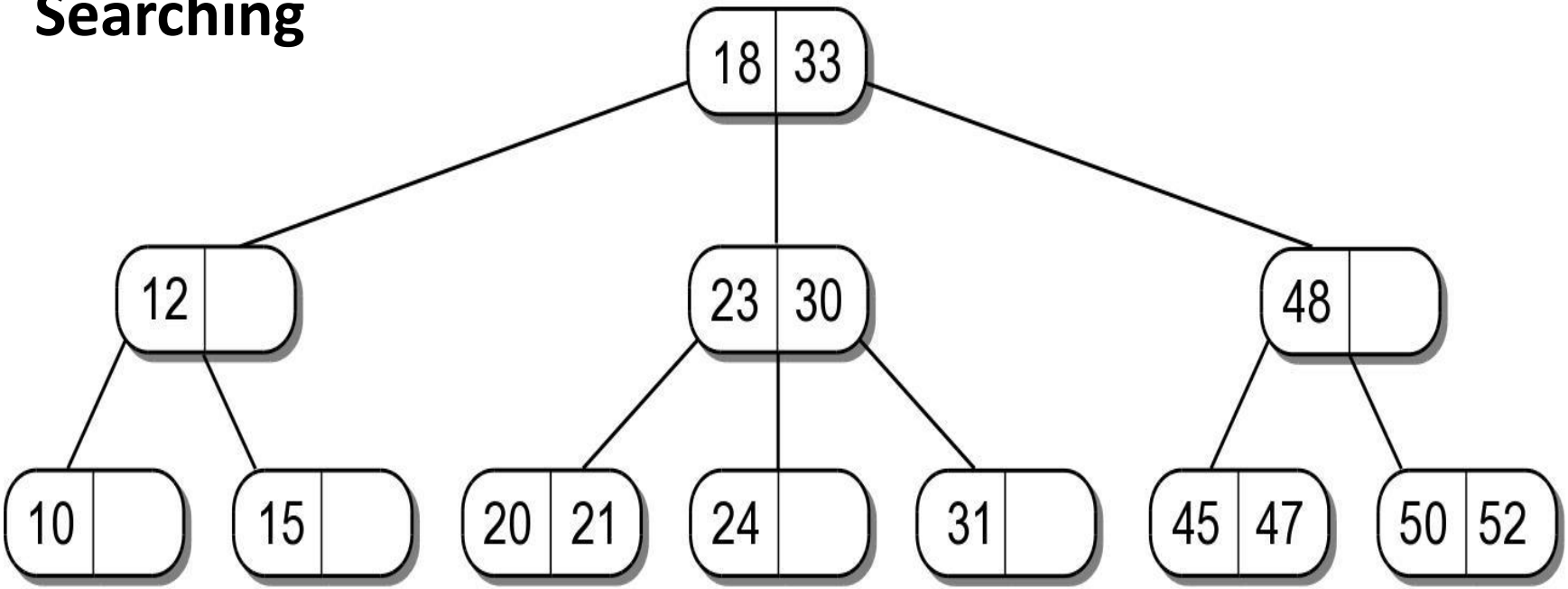
Properties

For an internal node with 2 keys and 3 children



Searching a 2-3 Tree

Searching



- *More cases*: empty tree, leaf, 2 children, 3 children.
- Easy to generalize from Binary Search Tree

Searching in a nonempty 2-3 tree

```
if target < node.key1:
  node = node.left                    # search in left subtree

if target == node.key1:
  return node                         # found it

if (node has two keys) and (target < node.key2):
  node = node.middle                  # search in middle subtree

if (node has two keys) and (target == node.key2):
  return node                         # found it

if (node has two keys) and (target > node.key2):
  none = node.right                   # search in right subtree
```

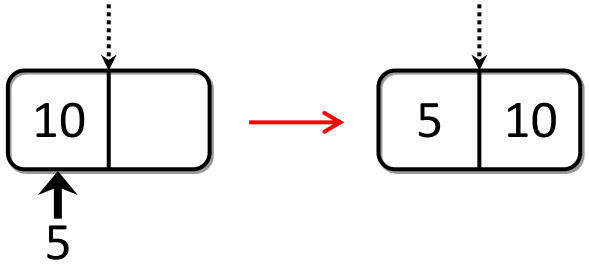
Inserting into a 2-3 tree

Key Idea of Inserting

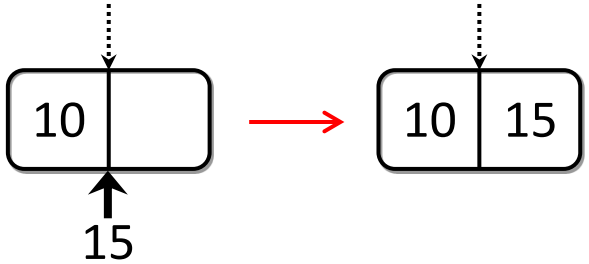
Know this key idea and you can handle any case.

if room

if new item lower, shift right

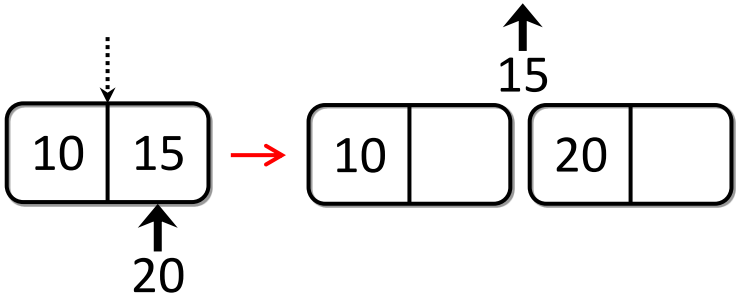


if new item higher, add right



if no room

keep low and high value, promote middle value and split in two



Inserting into a 2-3 tree

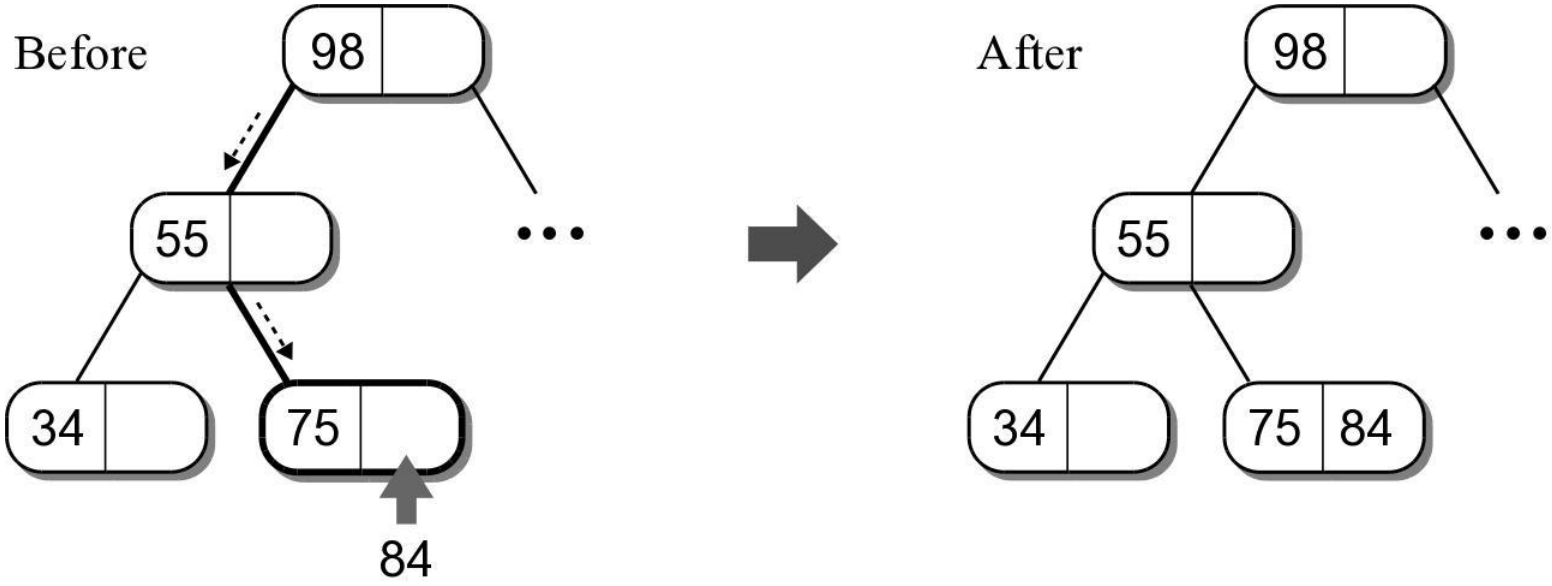
Basic Structure

- *Search* to find leaf where the new key should be
- More cases
  1. leaf has *1 key* → there's room for more
  2. leaf has *2 keys*→ make room for more
     a) Parent has 1 key
     b) Parent has 2 keys
     - If needed, *split* node and *promote* a key
     - *Changes may propagate* up the tree

Inserting into a 2-3 tree

Case 1: Leaf has 1 Key

- *Insert into a leaf* with room, e.g. insert 84

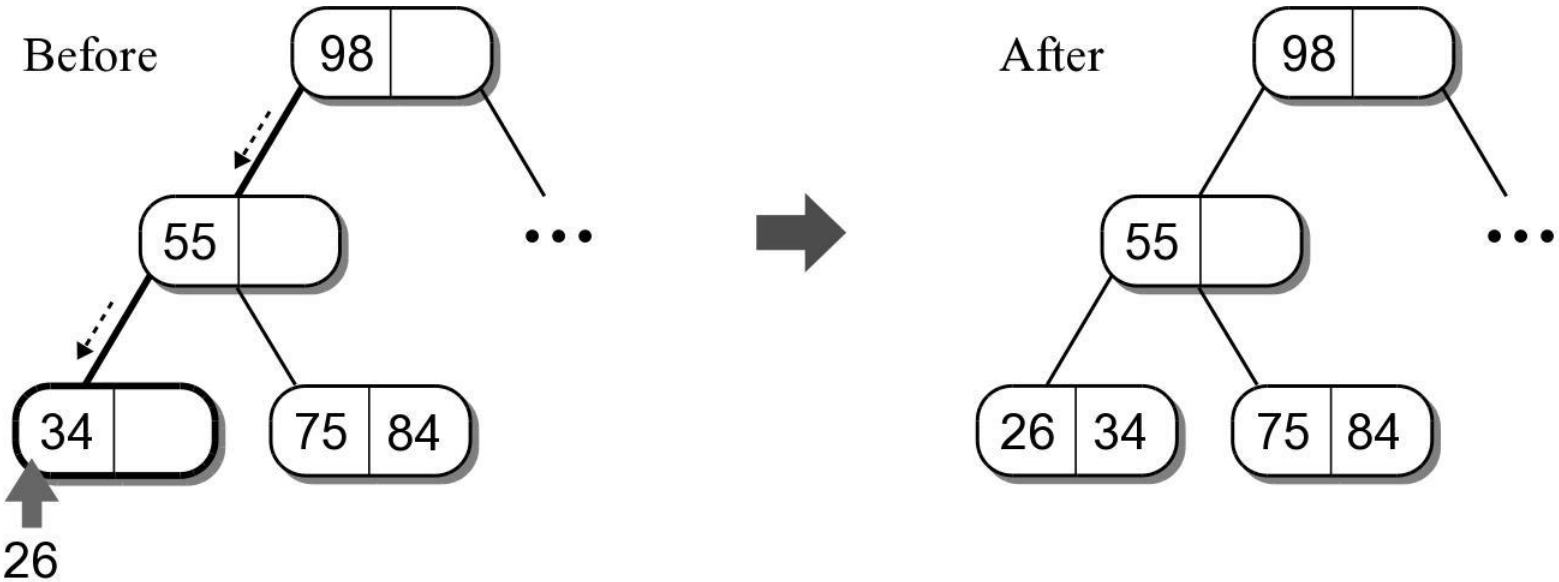


- *Key Idea: new item* (84) *higher → add right*

Inserting into a 2-3 tree

Case 1: Leaf has 1 Key

- *Insert into a leaf* with room but where some *shifting within the node* must take place, e.g. insert 26

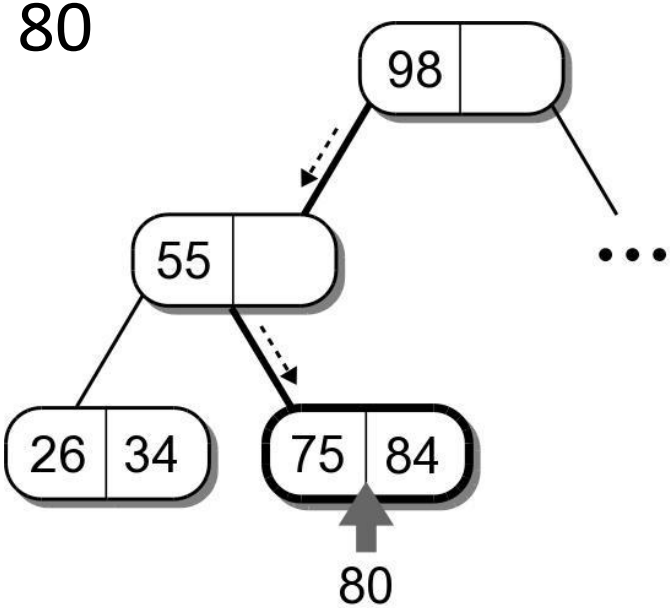


- *Key Idea: new item* (26) *lower → shift right*

Inserting into a 2-3 tree

Case 2: Leaf has 2 Keys

- E.g. insert 80



- *Key idea: node full → promote middle value*
- *Invariant*: All leaves must be at the same level in the tree – so we can't just add 80 as a leaf

Inserting into a 2-3 tree

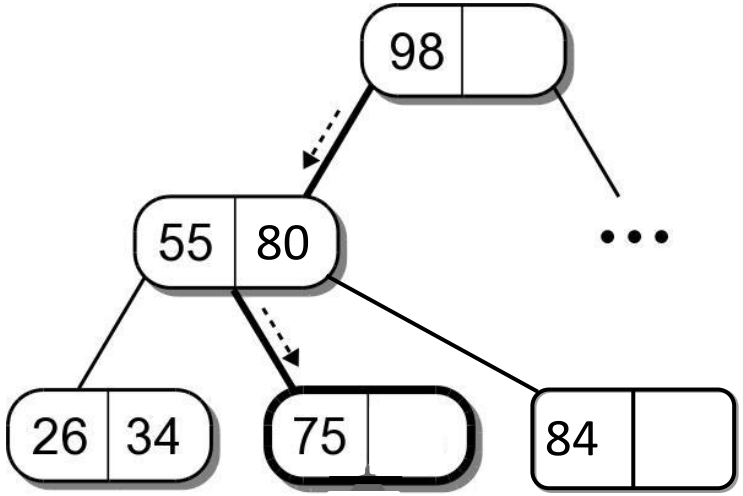
Case 2: Leaf has 2 Keys

- *Task:* Creating a new leaf
- Split the node (75|84) into two nodes:
  - *Replace* the single node with two nodes
  - *Put* the *smallest* key in one node: (75|)
  - *Put* the *largest* key in another node: (84|)
  - *Promote* the *middle* value (80) to the parent node
  - The two new nodes become children of the parent (replacing the original node)

Inserting into a 2-3 tree

Case 2: Leaf has 2 Keys

- must split the node
- E.g. insert 80



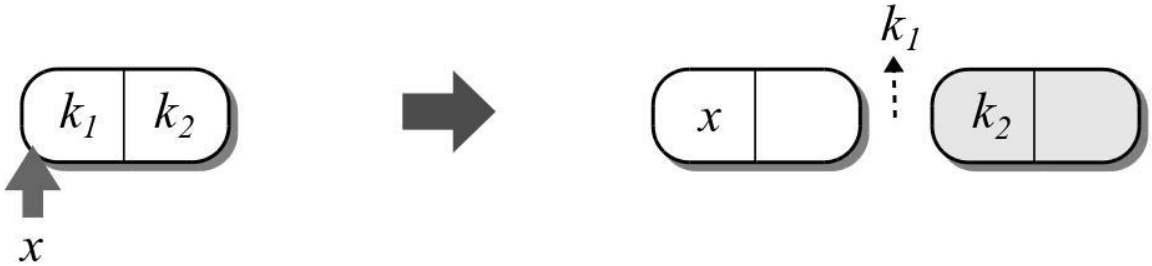
- Split (75|84) and promote middle value 80.

Inserting into a 2-3 tree

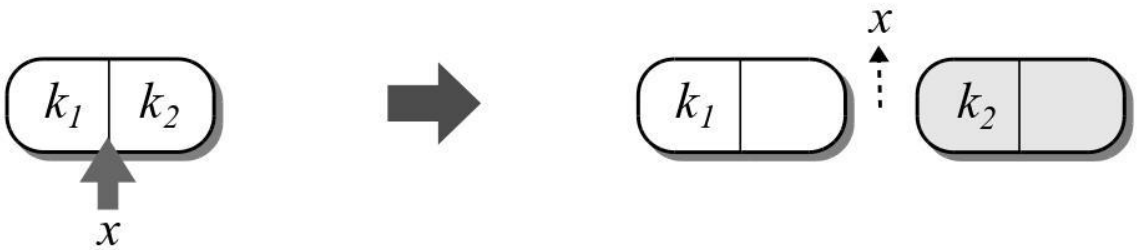
Case 2: Leaf has 2 Keys

Splitting when inserting *x*, *middle value gets promoted*

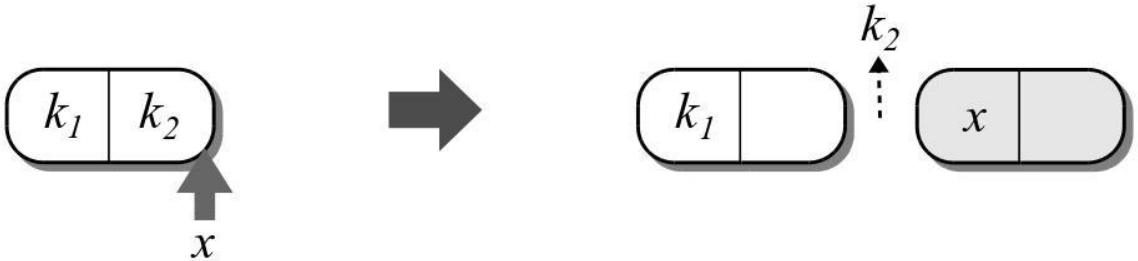
(a) x is the smallest key.



(b) x is the middle key.



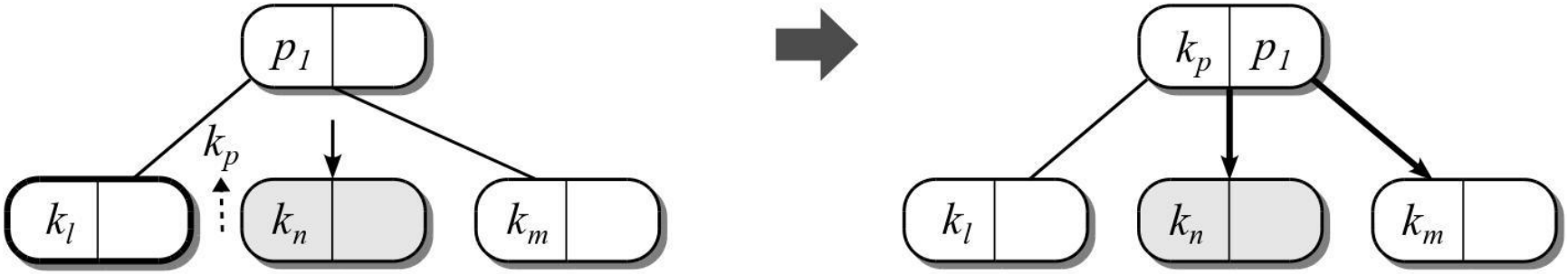
(c) x is the largest key.



Inserting into a 2-3 tree

Case 2a: Leaf has 2 Keys, Parent has 1 Key

(a) Splitting the left child.



When splitting the left child ...

k_p is coming from the *left child* so put it to the *left of* p_1

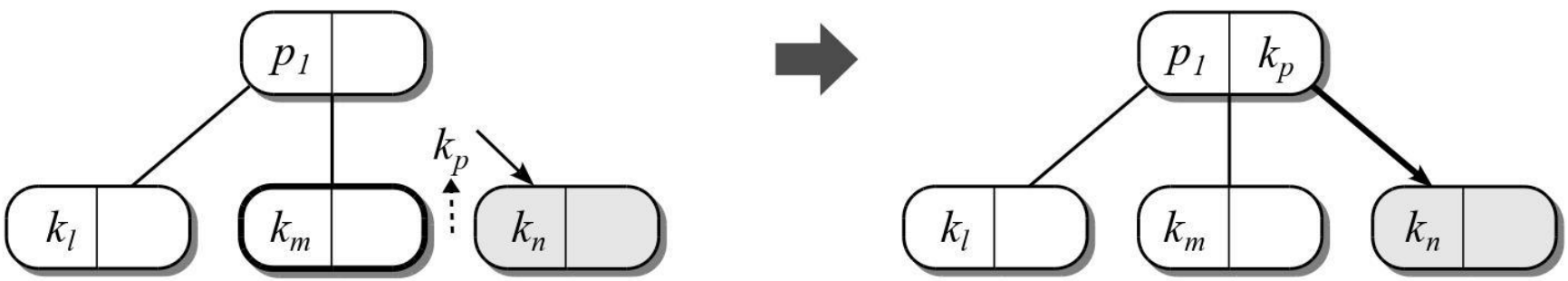
Inserting into a 2-3 tree

Case 2a: Leaf has 2 Keys, Parent has 1 Key

When splitting the middle child ...

k_p is coming from the *middle child* (which has values greater than p_1) so put it to the *right of p_1*

(b) Splitting the middle child.



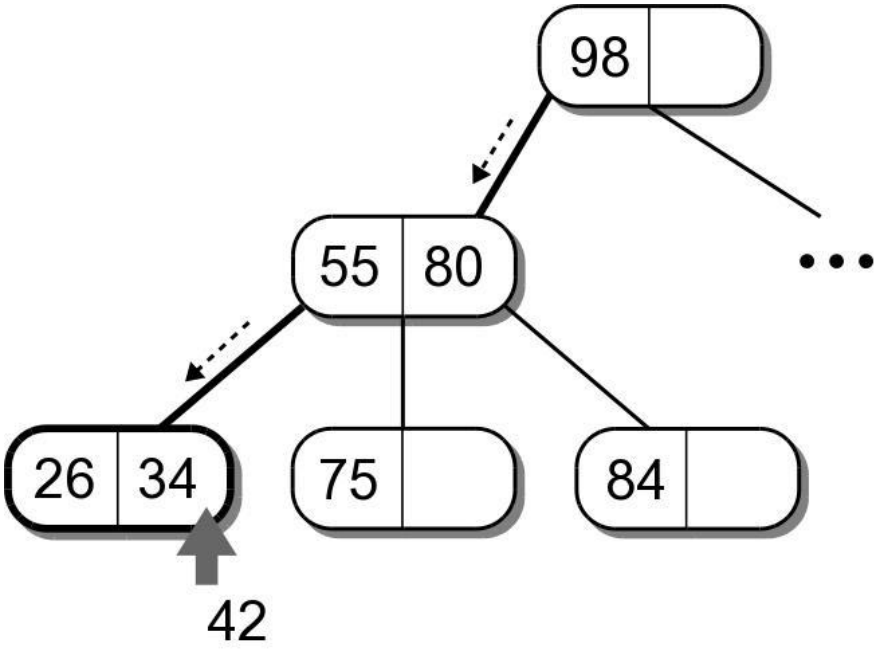
Inserting into a 2-3 tree

Case 2b: Leaf has 2 Keys, Parent has 2 Keys

- What if the parent is already full?
- Must *split the parent*

Example: inserting 42

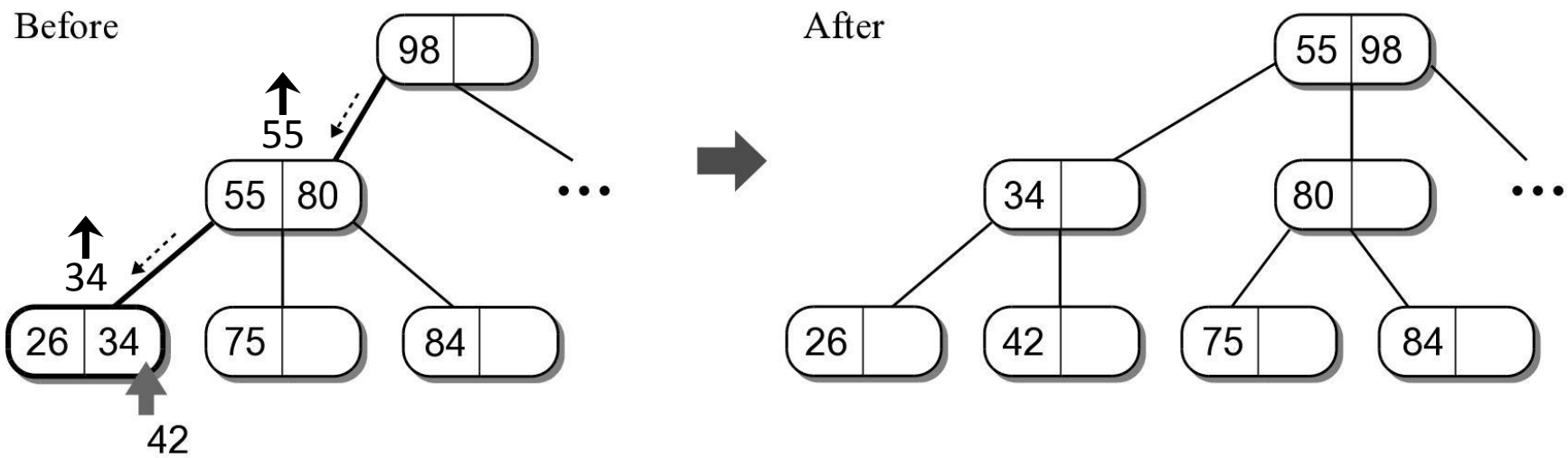
- search to the left of 98,
- then left of 55,
- then right of 34.
- This node is full and so is its parent



Inserting into a 2-3 tree

Case 2b: Leaf has 2 Keys, Parent has 2 Keys

- *Recursively repeat* what you would do with a leaf, i.e. middle gets promoted
- middle of 26, 34, 42 (i.e. 34) gets promoted
- middle of 34, 55, 80 (i.e. 55) gets promoted

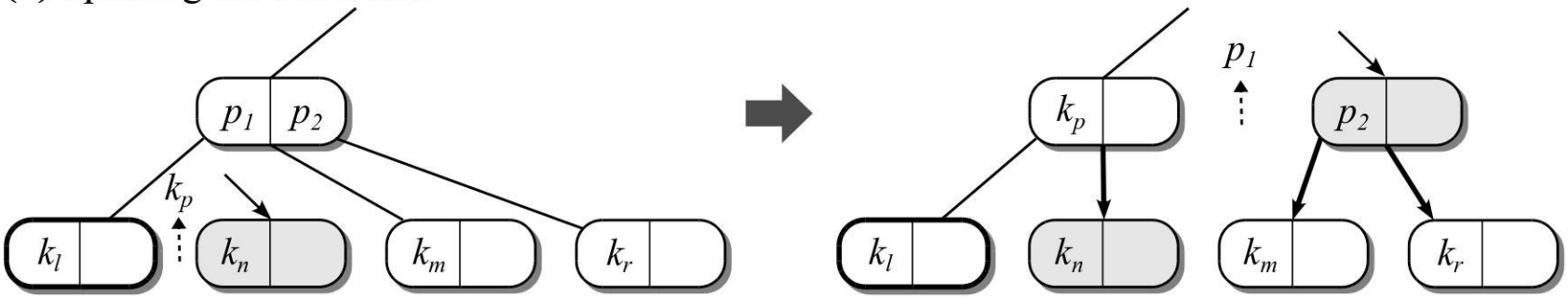


Inserting into a 2-3 tree

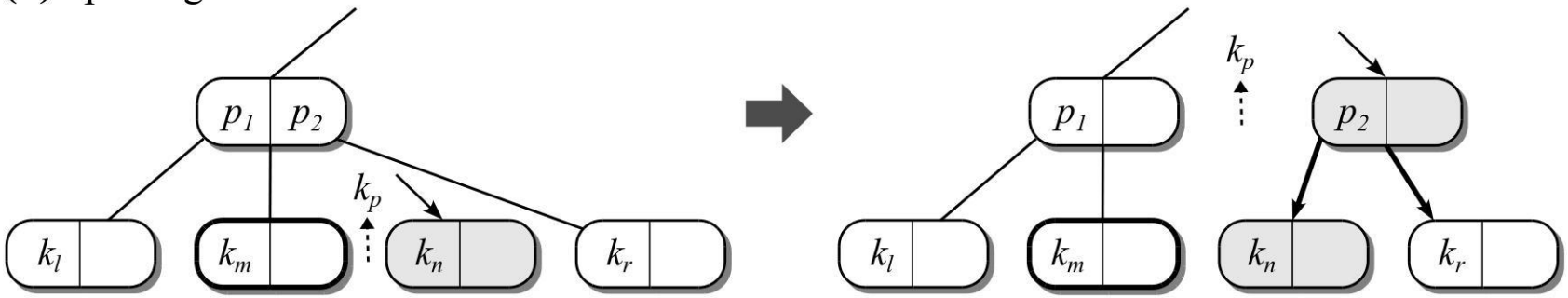
Case 2b: Leaf has 2 Keys, Parent has 2 Keys

- When the parent node is split, *a new parent node is created* at the same level.
- The keys (2 from the parent and 1 promoted) *have to be distributed*.
- *Links* from the parent to children must be *updated*.
- There are three cases, splitting the
  1. left child
  2. middle child
  3. right child

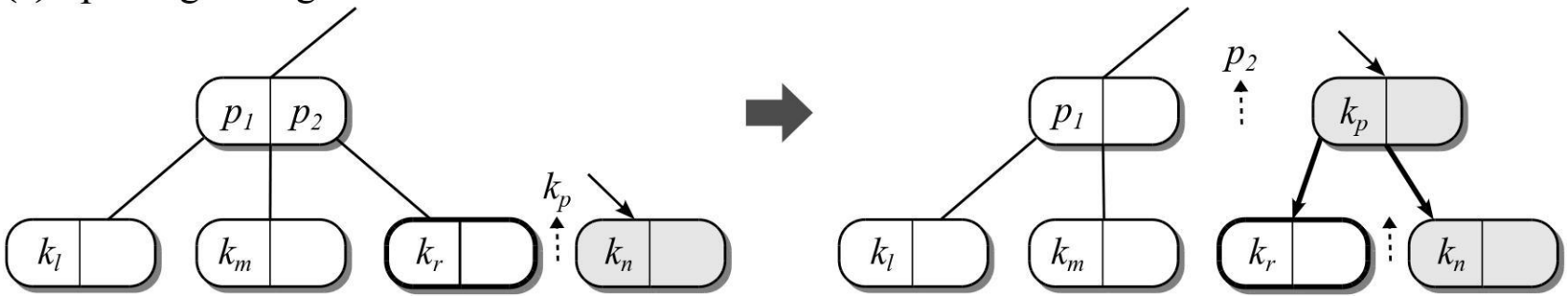
(a) Splitting the left child.



(b) Splitting the middle child.



(c) Splitting the right child.



Inserting into a 2-3 tree

Case 2b: Leaf has 2 Keys, Parent has 2 Keys

(a) splitting the left child

- k_p is from the *left child* so put it to the *left of* p_1
- $k_p < p_1 < p_2$ middle gets promoted: p_1

(b) splitting the middle child

- k_p is from the *middle child* so put it in the *middle*
- $p_1 < k_p < p_2$ middle gets promoted: k_p

(c) splitting the right child

- k_p is from the *right child* so put it to the *right of* p_2
- $p_1 < p_2 < k_p$ middle gets promoted: p_2

Inserting into a 2-3 tree

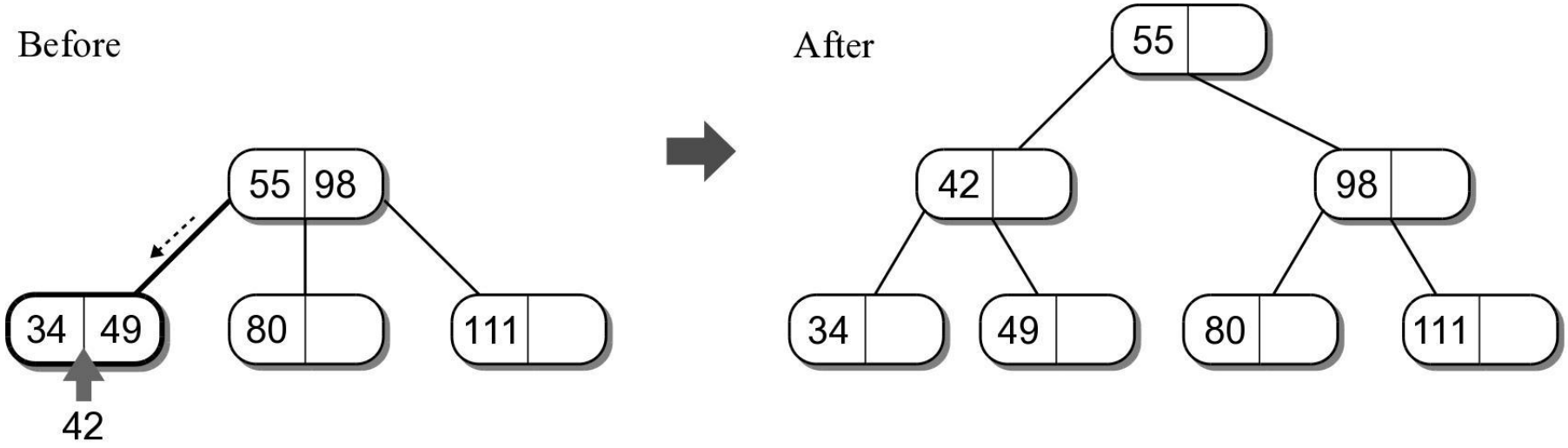
Completing the insertion

- Process is recursive
- The promotion may extend all the way up the tree
- We may have to split the root
- *But root has no parent!*

Inserting into a 2-3 tree

Splitting the root

- A new root is created
- With the new nodes as its children
- *Key idea: node (i.e. root) full → promote middle value*



Deleting from a 2-3 tree

Deleting

- Process is the opposite of insertion
- *Difference*: must ensure each *interior node has enough children*, rather than too many!
- *We won't cover it*

Efficiency of 2-3 Tree

Size

- assume the 2-3 tree has *n* keys

Searching

- 2-3 tree has between $\log_3 n$ and $\log_2 n$ levels
- it takes O(1) comparisons per level
- total: O(log *n*)

Efficiency of 2-3 Tree

Insertion

- Search to find location O(log *n*)
- Update the tree
  - make at most O(1) changes per level of the tree
  - there are O(log n) levels
- total: O(log *n*)

Deletions

- reverse of insert
- total: O(log *n*)

Mini Review: ADT Map

The Map ADT

- Consists of a collection of (key, value) pairs
- Each key is unique (think Python dictionary)
- Operations:
  - `Map()`
  - `length()`
  - `contains(key), valueOf(key)`
  - `add(key, value)`
  - `remove(key)`

Mini Review: ADT Map

Possible Implementations of Map

- Unordered Python list/array/SLL
  - Sequential search
  - Self-organizing lists: move to front, transpose, etc
  - Hash Tables :linear, quadratic probing, separate chaining
- Ordered Python list/array
  - Binary search
  - Interpolation search
- Search Trees
  - Binary search tree
  - 2-3 tree

ADT Map Implementation	Worst Case Runtime		
	contains	add	remove
Unordered List			
Sequential Search	O(n)	O(n)	O(n)
Self-organizing	O(n)	O(n)	O(n)
Hash Table (probe)	O(n)	O(n)	O(n)
Hash Tables (sep. chain)	O(n)	O(n)	O(n)
Ordered array			
Binary Search	O(log n)	O(n)	O(n)
Interp. Search	O(n)	O(n)	O(n)
Search Trees			
BST	O(n)	O(n)	O(n)
2-3	O(log n)	O(log n)	O(log n)

ADT Map Implementation	If all goes according to plan …		
	contains	add	remove
Unordered List			
Sequential Search	O(n)	O(1)	O(n)
Self-organizing	O(n)	O(n)	O(n)
Hash Table (probe)	O(1)	O(1)	O(1)
Hash Tables (sep. chain)	O(1)	O(1)	O(1)
Ordered array			
Binary Search	O(log n)	O(n)	O(n)
Interp. Search	O(log log n)	O(n)	O(n)
Search Trees			
BST	O(log n)	O(log n)	O(log n)
2-3	O(log n)	O(log n)	O(log n)

Topic 15: B-Trees

Key Concepts

- not in text

Recall

- Topic 14.4 2-3 Trees

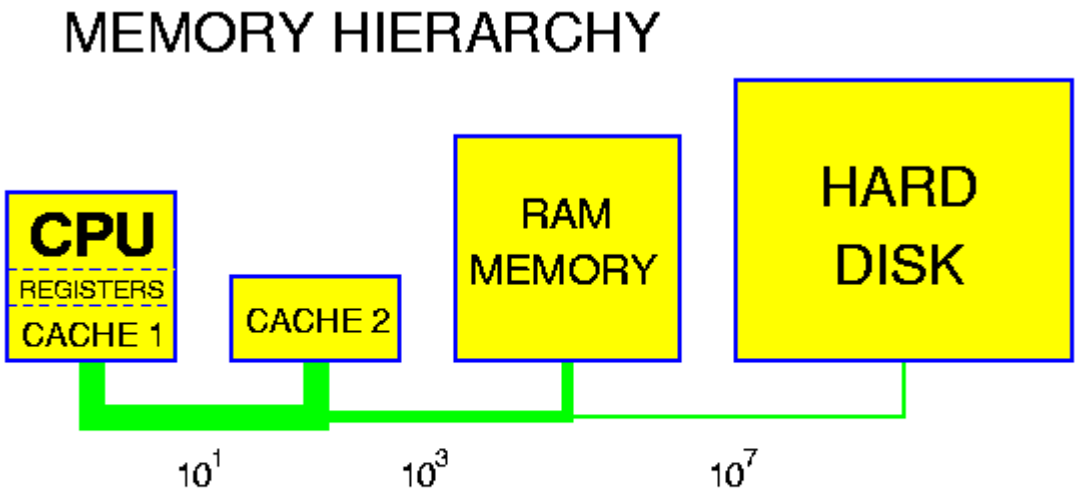
B-Trees Motivation

Motivation

- We've been ignoring some very important issues in our algorithms
- *Accessing memory*
  - We've assumed each memory access takes about the same amount of time
  - That is not always the case
- We need to know (a little) more about how memory is organized

B-Trees Motivation

Faster, smaller
more expensive



Indicated are approximate numbers of clock cycles to access the various elements of the memory hierarchy

Slower, bigger,
less expensive

There is a hierarchy of how fast data can be moved from memory

Source: http://users.informatik.uni-halle.de/~jopsi/dinf104/chap3.shtml

B-Trees Motivation

Limitations

- Ideally, everything would fit into the CPU registers.
  - Not likely, those are relatively small.
- Instead, data will be stored at the hierarchy level large enough to hold it.
- Differences in access time can be extreme, comparing cache access to hard drive access, to external disks, sticks, CDs, DVDs, etc.

B-Trees Motivation

Locality of Reference

Most software exhibits...

- *Temporal locality*

  Software usually uses a small amount of data over and over again (e.g. counters, local variables, 80/20 rule)

- *Spatial locality*

  Software accessing one piece of data often accesses that data's neighbours (e.g. array entries, reading in a file)

B-Trees Motivation

Caching

- Inspired by *temporal locality*
- When data is accessed in *secondary memory* (slower in the memory hierarchy), bring it into *primary memory* (faster in the memory hierarchy)
- If accessed again, it will be faster

B-Trees Motivation

Blocking

- Inspired by *spatial locality*
- When data at secondary-level (i.e. slow) memory location L is accessed, *bring in a block of data* that includes L into primary-level (i.e. faster) memory
- For next memory access, if a neighbour is accessed, it will be faster

B-Trees Motivation

Resulting Problem

- There is a limit to how many blocks can fit into primary-level memory
- We may have to swap blocks as the program proceeds

  How to choose which to remove?
- Algorithms can be designed to limit the number of block swaps needed
- *B-trees* (*a generalization of 2-3 trees*) addresses this problem

B-trees

B-Tree Structure

- Balanced: All leaves are at the same level
- Each node has: (for odd *d, so there is always a middle*)
  - between $\left\lfloor \frac{d}{2} \right\rfloor$ and *d-1* keys
  - one more child than it has keys
- Except:
  - Leaf nodes – no children
  - Root can be a leaf (with up to d-1 keys), or can have as few as 1 key with 2 children

B-trees

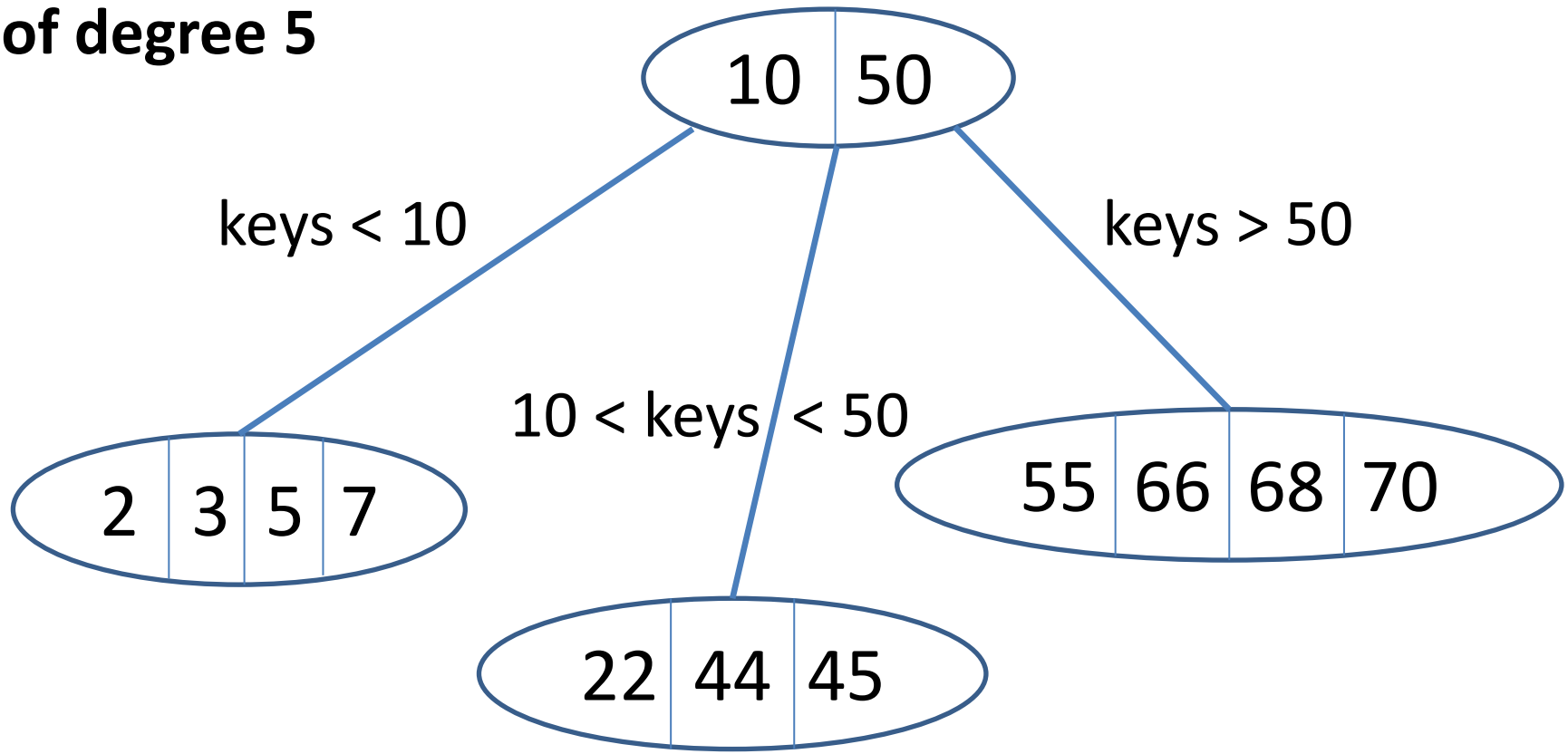
B-tree Structure

- *Again: similar to 2-3 tree but with many more keys*
- Keys are kept in increasing order:

  $k_1 < k_2 < k_3 < \ldots k_m$
- Keys in the *left-most* subtree are *less than* k_1
- Keys in the *right-most* subtree are *greater than* k_m
- Keys in a middle subtree *s* satisfy: $k_{s-1} <$ *keys* $< k_s$

B-tree of degree 5

Example of a B-tree of degree 5



Searching for target in a B-tree

Search

- *Modify search for 2-3 trees to handle more keys*
- Start at the root
- Search keys in the root
  - Is target one of these keys?
  - If not, determine which subtree to search in
- *If* empty subtree found *then* target is not there

Efficiency of Searching in a B-tree

Efficiency

- B-trees are balanced trees
- Size: *n* nodes
  - height is $O(\log_d n)$
  - all operations (search, add, remove) are $O(\log n)$

Inserting a key into a B-tree

Insert

- *Modify Insert for 2-3 trees to handle more keys*
- Find leaf which should contain new key
- If room, place key there
- If not, split the node into two pieces
  - Place smallest keys in one node
  - Place largest keys in other node
  - Promote median to parent
- Continue until parent has room or the root is split

B-Trees and the Memory Hierarchy

Using the Memory Hierarchy

Question: how do B-trees help reduce swapping blocks in memory?

Answer: Choose degree *d* to be the largest odd integer such that *d* references and *d*-1 objects fit into a single *block* .

I.e. the size of the *block* that is read into primary memory to take care of *locality of reference*

B-Trees and the Memory Hierarchy

Using the Memory Hierarchy

Question: how do B-trees help reduce swapping blocks in memory? cont...

- Let B be the number of items that can fit in one block
- Degree *d* is $\Theta(B)$
- *Number of Retrievals* will be $O(\log_B n)$
- *Number of block swaps* will be $O(\log_B n)$

B-Trees and the Memory Hierarchy

Using the Memory Hierarchy

Question: how do B-trees help reduce swapping blocks in memory? cont...

- Consider a dictionary with 4,228,250,625 keys, B=255
- The height of the tree is $\log_{255}(4{,}228{,}250{,}625) = 4$
- *Number of Retrievals* will be $O(\log_B n) = 4$
- *Number of block swaps* will be $O(\log_B n) = 4$
- Searching through 255 keys takes time, but it will be *quicker* than reading in a new block of data.

B-Trees and the Memory Hierarchy

System Considerations

- Just 2-3 trees with a more keys and children
- 2-3 trees modified to handle large data sets
- Can be specialized to known system parameters
- On a different system, a different degree d may be needed for the tree
- typically used in databases to get data quickly

Topic 16: Graphs

Key Concepts

- the Graph ADT
- representing graphs
- traversing graphs
- graph terminology

Recall

- Topic 10 Binary Trees (slides 201-212)
- particularly: traversing binary trees, depth first search and breadth first search (slides 208-212)

Graphs

What do the following problems have in common?

Traveling Salesman Problem: Given a list of cities and the distances between them, find the shortest tour that visits all cities and returns to the start.

Graphs

What do the following problems have in common?

Resource Allocation: Given the changing demographics of a city, where should a new fire station be built so all residents have equal access?

Graphs

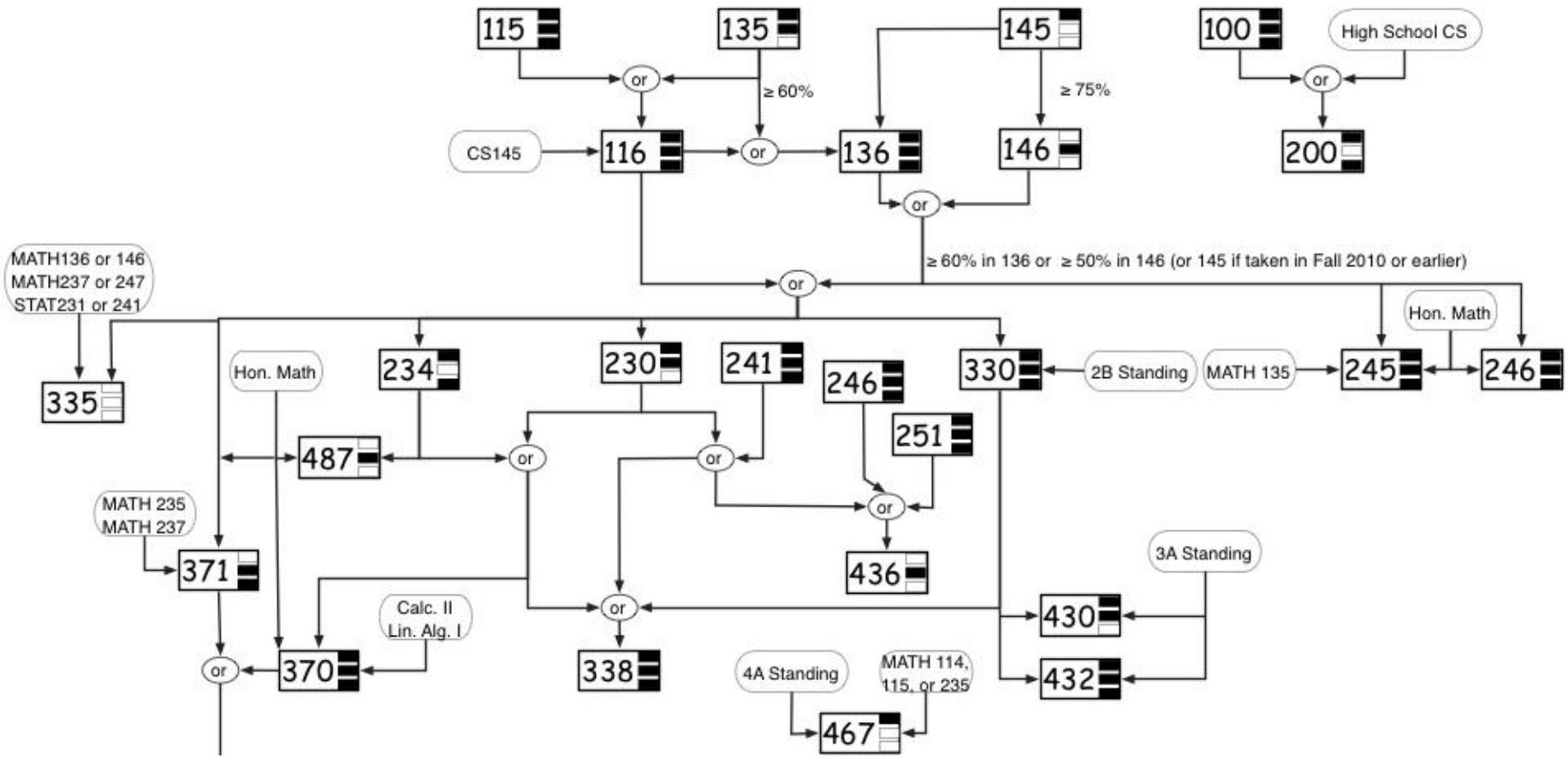
What do the following problems have in common?

Web crawler (data collecting part of a search engine): How best to find web sites and the links between them.

Graphs

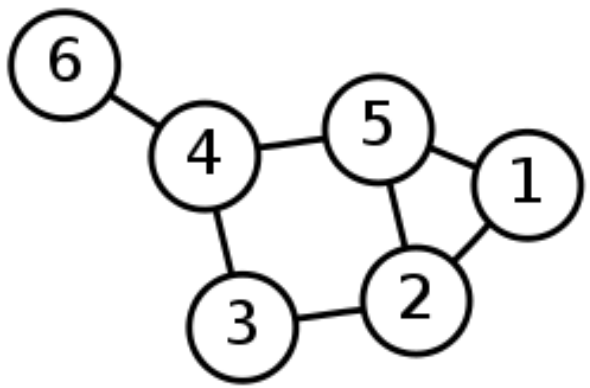
What do the following problems have in common?

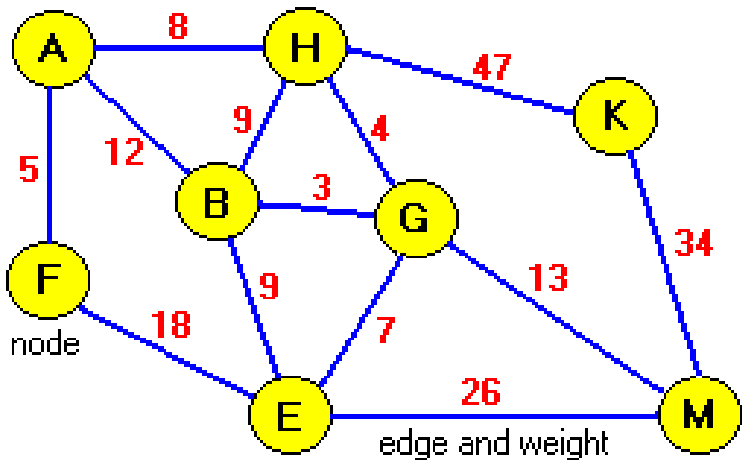
Course Enrollment: How to check course pre-requisites.



Graphs

Definition: An *undirected graph* $G(V, E)$ is a set V of *vertices*, and a collection E of unordered pairs from V, called *edges*. Do not allow self-loops.





Terminology: adjacent (neighbour), degree (incident), label, path, connected, weight.

Graphs

Caution. There are *many types* of graphs and it is not always obvious what assumptions are being made, so be careful about what you read (e.g. on the web) because what may be *true for one type* of graph may *not be true for another type.*

Graphs can be:

- cyclic vs. acyclic
- directed vs. undirected
- connected vs. component
- allow self-loops vs. no self-loops

For Example

What is the maximum number of edges in a graph with *n* vertices?

The Graph ADT

Operations on Graphs

- Graph(V, E): create a new graph
- isEmpty(): does G have any vertices
- numEdges(): how many edges in G
- numVertices(): how many vertices in G
- edges(): return a list of all the edges in G
- vertices(): return a list of all the vertices in G
- isAdjacent(v1,v2): is {v1, v2} an edge in G
- path(v1,v2): is there a path from v1 to v2

The Graph ADT

Operations on Graphs

- addVertex(v): add a vertex to G
- removeVertex(v): remove *v* and the edges that contain it
- addEdge(v1,v2): add the edge {v1, v2} to G
- removeEdge(v1,v2): remove edge but leave vertices
- neighbours(v): return a list of all vertices adjacent to *v*
- traverse(v): return a list of vertices connected to *v*
- isConnected(): is G a connected graph

Graphs are a rich data type (*many* possible operations)

The Graph ADT

Using the Graph ADT

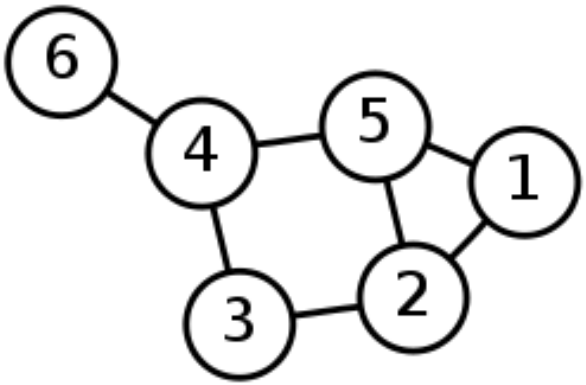
What is this snippet of code doing?

```
for v1 in G.vertices():
  for v2 in G.vertices():
    if (v1 != v2) and G.path(v1,v2) == False:
      G.addEdge(v1,v2)
```

Graph Implementations

Implementation 1: Edge List

Represent G as a list of edges [e_1, e_2, e_3, …, e_m], where each edge e_j is a pair of vertices = (v_{j1}, v_{j2})



Example

Edge list = [(4, 6), (4,5), (3, 4), (3,2), (2,5), (1,2), (1, 5)]

Graph Implementations

Implementation 2: Adjacency Lists

For each vertex, list the vertices adjacent to it.

Example

1: [2, 5]

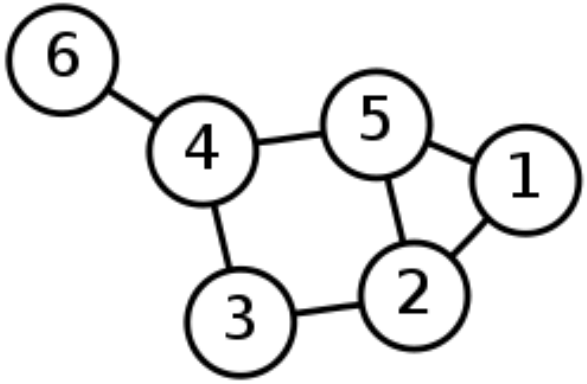
2: [1, 3, 5]

3: [2, 4]

4: [3, 5, 6]

5: [1, 2, 4]

6: [4]



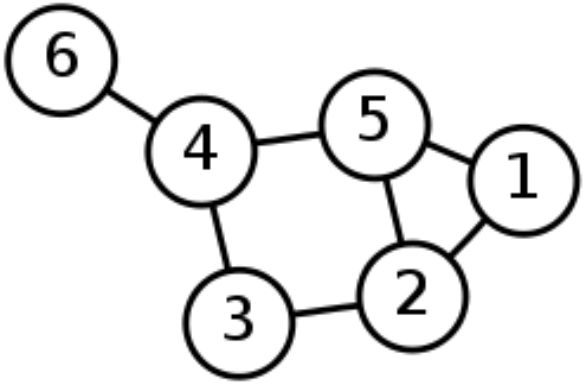
Graph Implementations

Implementation 3: Adjacency Matrix

- Create an *n* x *n* matrix where

  if $\{v_i, v_j\}$ is an edge *then* both m[i-1, j-1] and m[j-1, i-1] are 1, *else* both are 0.

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



Graph Implementations

Weighted Graphs

Edge List:

- store weight with edge

Adjacency List:

- store weight with neighbour in list (vertex, weight)

Adjacency Matrix:

- store weight in matrix at location for v_i, v_j
- need special value, None, to distinguish zero weight from no edge

Graph Implementations

Adjacency List Implementation

- Use a *dictionary* { v:adj(v) }, where
  - *keys*: *v* are the labels on vertices
  - *values*: adj(*v*) is a list (SLL or Python list) of vertices adjacent to *v*
  - E.g. *G* = {1: [2, 5], 2: [1, 3, 5], 3: [2, 4], 4: [3, 5, 6], ... }
- Use a *list of list* when the labels are non-negative ints
  - *index*: labels on vertices
  - *values*: list (SLL or Python list) of neighbours
  - E.g. *G* = [[2, 5], [1, 3, 5], [2, 4], [3, 5, 6], ...]

Graph Implementations

Adjacency Matrix Implementation

If vertices are labelled with *strings*

- use a *Dictionary* to map labels onto *ints*
  - entry is {*label*: *index*} where
  - *label* is the vertex label, and
  - *index* is the row/column index for that vertex
- Examples
  - cities = { 'KW':0, 'Cambridge':1, 'Guelph':2, ... }
  - courses = { 'CS115':0, 'CS116':1, 'CS135':2, ... }

Graph Implementations

Adjacency Matrix Implementation

Assume: vertices are now labelled with ints 0, 1, 2..

- *Numpy* special package for dealing with matrices
- *List of List*
  - *index*: *i* is the vertex label
  - *values*: AM[i] is the *i* th row of the adjacency matrix
  - AM[i][j] is the *j*th entry in the *i* th row of the matrix
- *Single List*
  - to represent an *n* x *n* matrix as a single python list, SPL, map matrix entry (*i*, *j*) to SPL[*n***i* + *j*]

Graph Traversals

Goal

Determine which vertices in G *can be reached* from a starting vertex *v.*

Note

- Multiple traversals possible
- Loops are possible: must mark vertices to ensure we do not loop indefinitely

Types of Graphs

- If all vertices can be reached, G is *connected*
- If not, a *connected component containing v* has been found

Depth-first Traversal

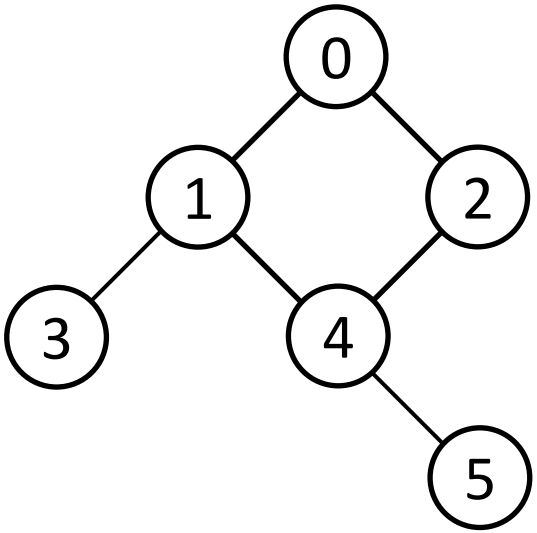
Basic Structure

- Need a starting vertex *v* (mark it)
- Recall slides 207-210 (trees are easier, start at root)
- Proceed along a path from *v* as far as possible
- Then, backup to previous vertex, and visit its unvisited neighbours
- Repeat while unvisited, reachable vertices remain
- Could be implemented recursively or with a stack.

Depth-first Traversal

Pseudocode

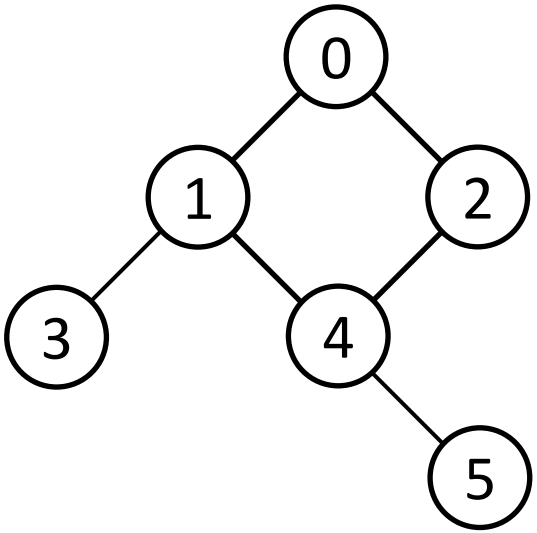
```
discovered = {}   # who's been discovered
s = Stack()   # who is waiting to processed
def dft(u):
  s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```



Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

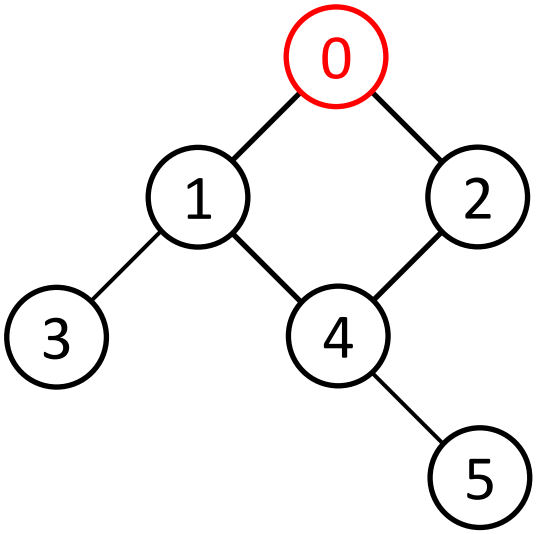


s = [0]
discovered.keys() = [0]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

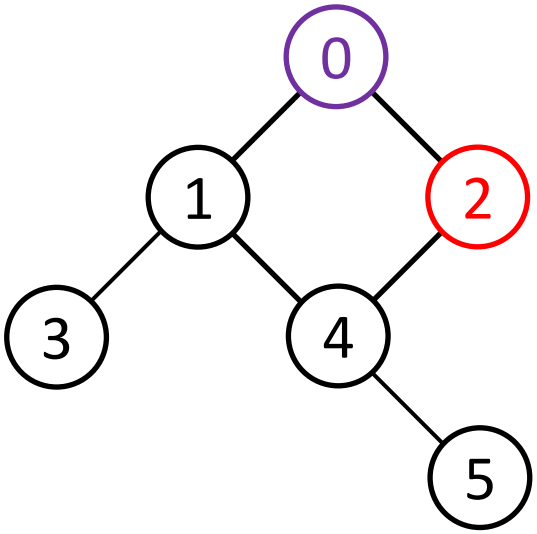


processed: 0
s = [1, 2]
discovered.keys() = [0, 1, 2]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

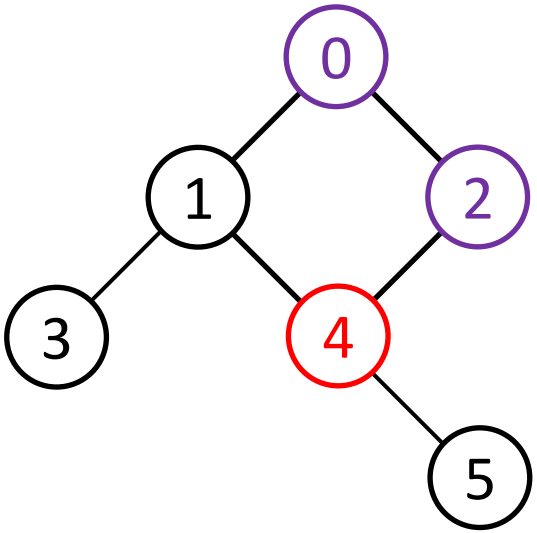


processed: 2
s = [1, 4]
discovered.keys() = [0, 1, 2, 4]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

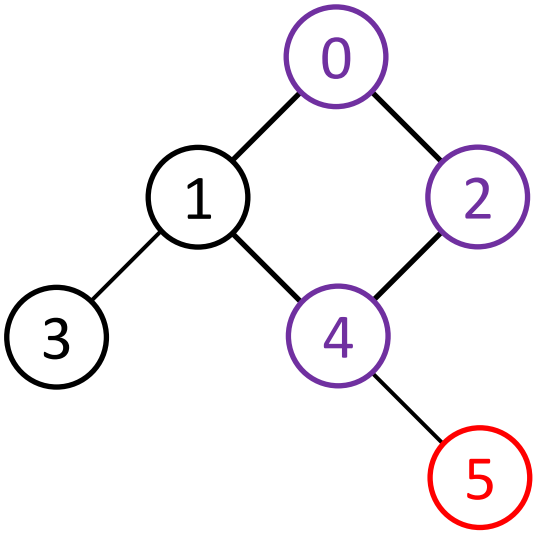


processed: 4
s = [1, 5]
discovered.keys() = [0, 1, 2, 4, 5]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

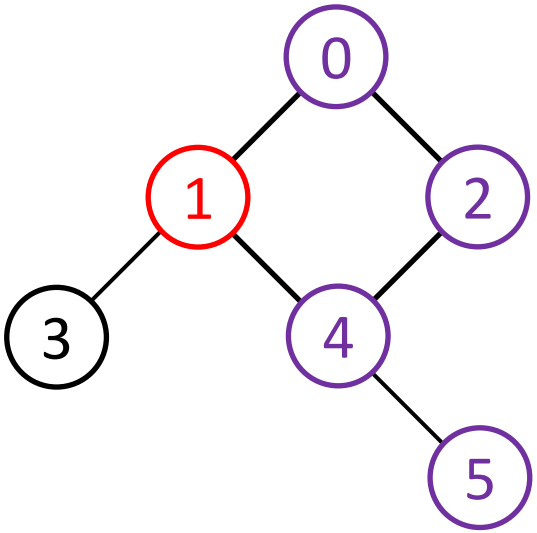


processed: 5
s = [1]
discovered.keys() = [0, 1, 2, 4, 5]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

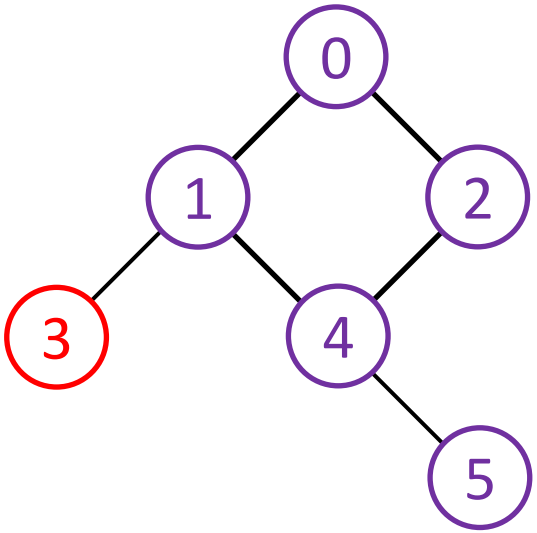


processed: 1
s = [3]
discovered.keys() = [0, 1, 2, 4, 5, 3]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```

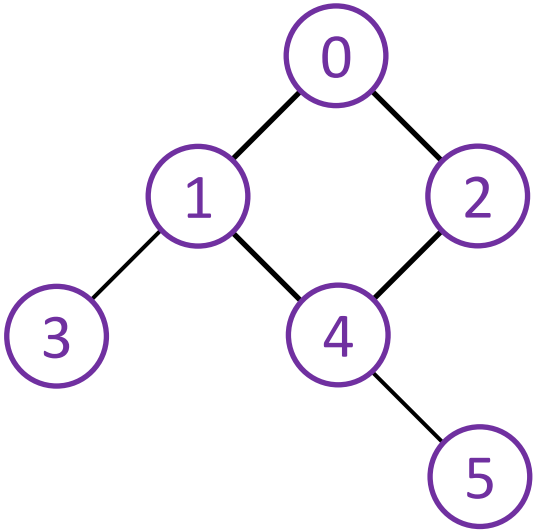


processed: 3
s = []
discovered.keys() = [0, 1, 2, 4, 5, 3]

Depth-first Traversal

Pseudocode

```
dft(0):
   s.push(u)
  discovered[u] = True
  while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
      if w not in discovered:
        s.push(w)
        discovered[w] = True
```



s = [] done!

discovered.keys() = [0, 1, 2, 4, 5, 3]

Depth-first Traversal

Applications

- Find all the vertices connected to v
- Determine if G is a connected graph
- Determine if there is a path between vertices *u* and *v*

Depth-first Traversal

Asymptotic Time Complexity

- *Assume:* there are *n* vertices and *m* edges
- At most *n* visits (each vertex visited once)
- Each visit examines neighbours of a vertex
  - At most 2*m* neighbours examined over all recursive calls (each edge has two endpoints)
  - To examine each edge, O(1) steps
- *Total is O(n+m)*

Breadth-first Traversal

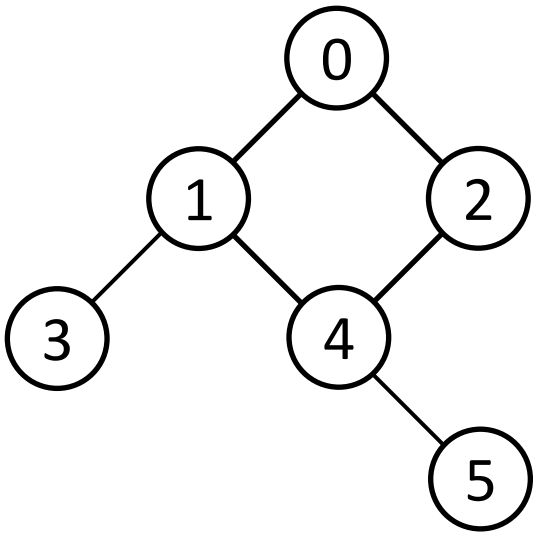
Basic Structure

- Recall slides 211 - 212 (trees are easier, start at root)
- Need a starting point *v* (mark it as discovered)
- Visit all the neighbours of *v*, marking them as you go
- Then, visit all of the neighbours of the neighbours of v, marking those in turn, etc.
- Repeat until all reachable vertices are marked
- Use a *queue* rather than a *stack*

Breadth-first Traversal

Pseudocode

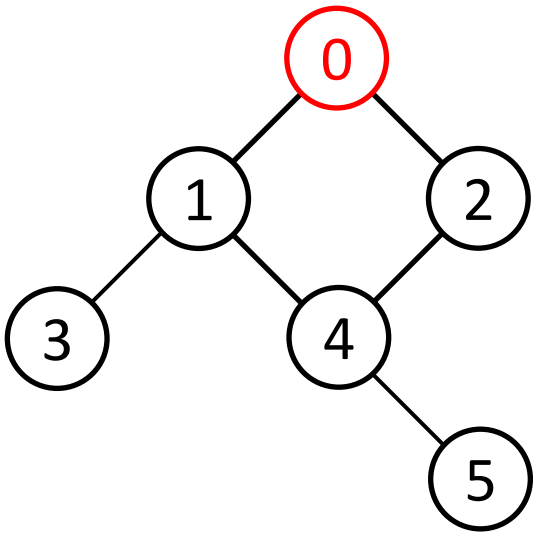
```
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v) # i.e. do what you want
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



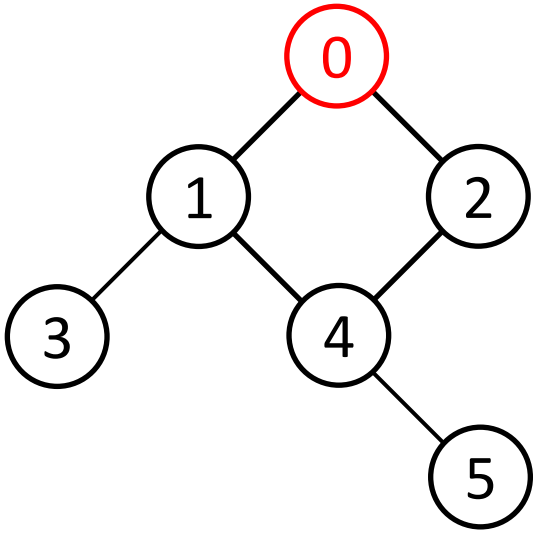
Q = [0]

discovered.keys: [0]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



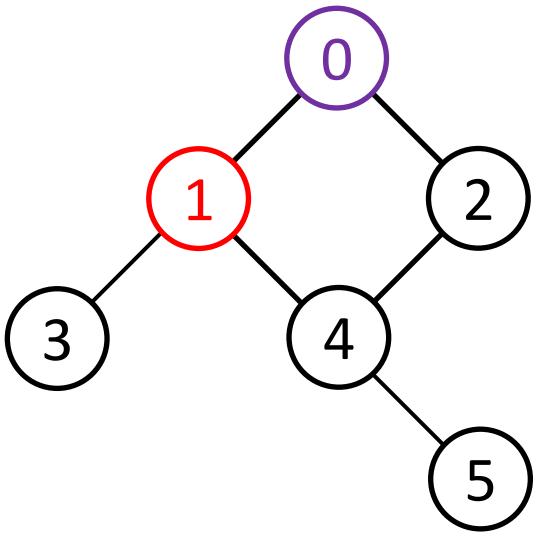
Q = [1, 2]

discovered.keys: [0, 1, 2]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



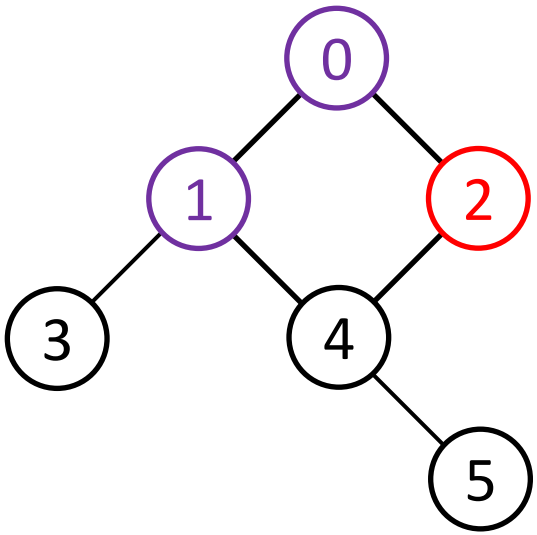
Q = [2, 3, 4]

discovered.keys: [0, 1, 2, 3, 4]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



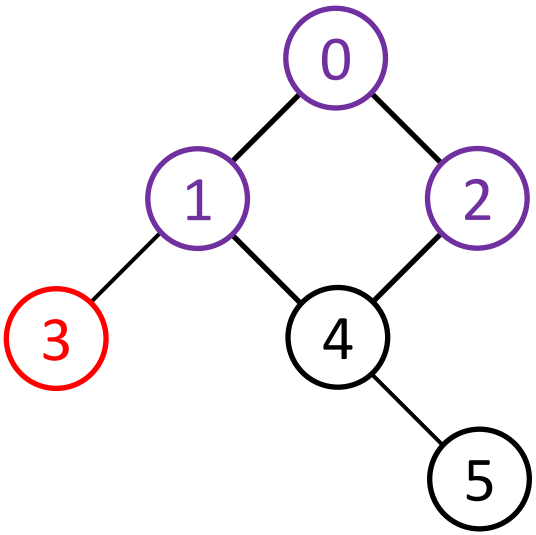
Q = [3, 4]

discovered.keys: [0, 1, 2, 3, 4]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



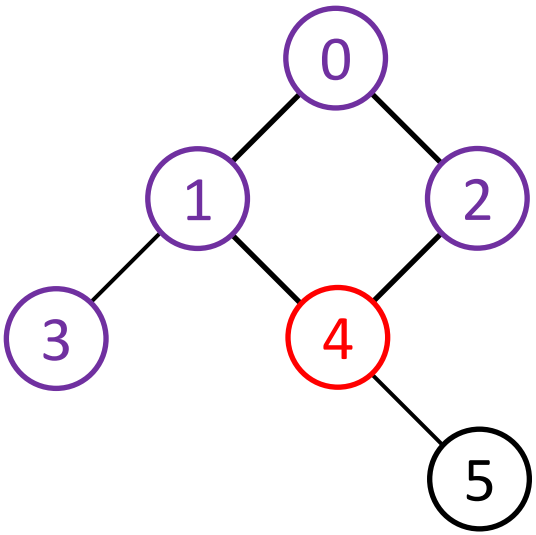
Q = [4]

discovered.keys: [0, 1, 2, 3, 4]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



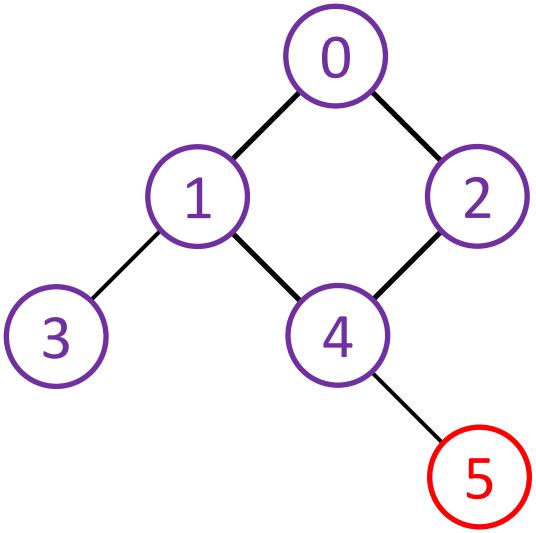
Q = [5]

discovered.keys: [0, 1, 2, 3, 4, 5]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



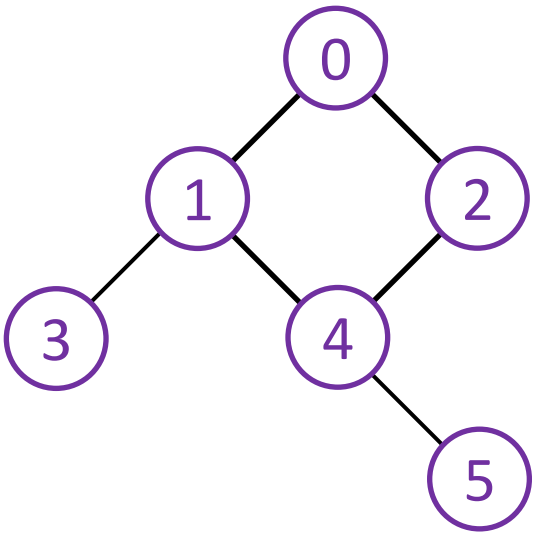
Q = []

discovered.keys: [0, 1, 2, 3, 4, 5]

Breadth-first Traversal

Pseudocode

```
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```



Q = []

discovered.keys: [0, 1, 2, 3, 4, 5]

Breadth-first Traversal

Applications

- Find all the vertices connected to *v*
- Determine if G is a connected graph
- Determine if there is a path between vertices *u* and *v*
- Find the shortest path from *v* to *w*

Breadth-first Traversal

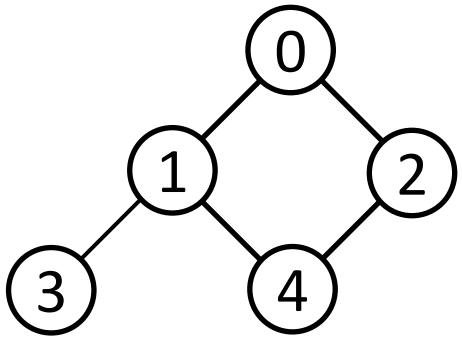
Asymptotic Time Complexity

- *Assume:* there are *n* vertices and *m* edges
- Examine all *n* vertices
  - Add to queue and mark as discovered both O(1)
  - O(n) steps
- Examine all *m* edges
  - check if the neighbours have been discovered, O(1)
  - O(m) steps
- *Total O(n+m),* same as Depth-first Traversal

Terminology

Vertices and Edges

- A graph G consists of a *set of vertices V* (also called nodes) and a *set of edges E*. In the graph below the *vertices* are {0, 1, 2, 3, 4} and the *edges* are described using pairs of *vertices*, i.e.{ {0, 1}, {0, 2}, {1,3}, {1,4}, {2,4} }.



G = (*V*, *E*)

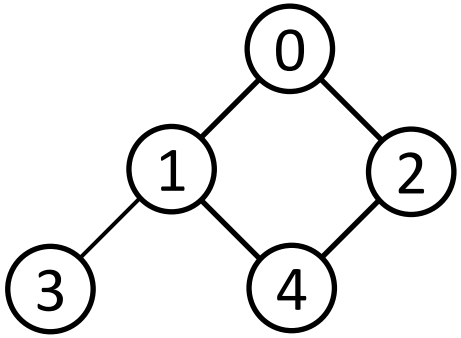
V = {0, 1, 2, 3, 4}

E = { {0,1}, {0,2}, {1,3}, {1,4}, {2,4} }

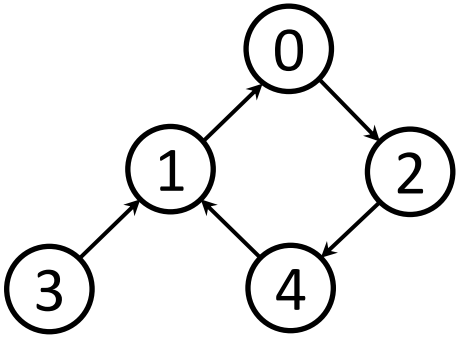
Terminology

Undirected Graphs

- An *undirected graph* is a graph such that if (u, v) is an *edge* in the graph then (v, u) is also an *edge* in the graph. Informally, in *undirected graphs* the *edges* do not have arrows indicating direction.



An Undirected Graph



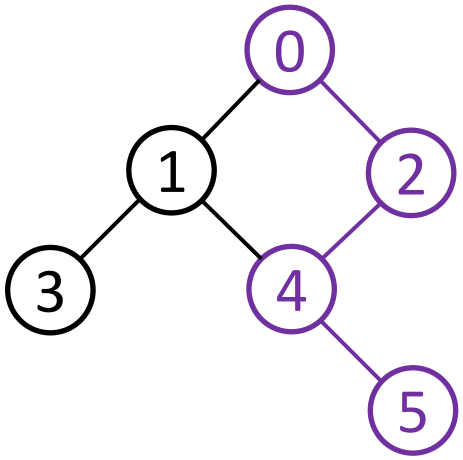
A Directed Graph

- We will be investigating *undirected graphs*.

Terminology

A Path

- A *path* from *vertex* v_1 to *vertex* v_k is a *sequence of vertices* $(v_1, v_2, ..., v_k)$ such that each pair $\{v_i, v_{i+1}\}$ in the sequence is an *edge* in the graph.

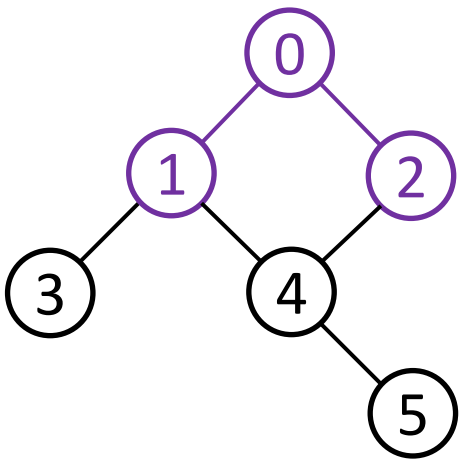


- (0, 2, 4, 5) is a *path* from *vertex* 0 to *vertex* 5.
- {0, 2}, {2,4}, and {4,5} are all *edges* in the graph

Terminology

Adjacent Vertices

- Two *vertices*, *u* and *v*, are *adjacent* if {*u*, *v*} is an *edge* in the graph (i.e. they are directly connected).



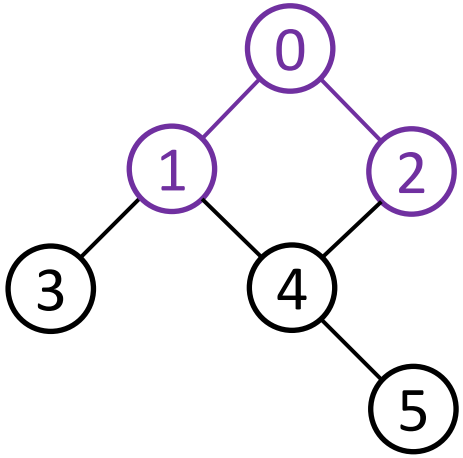
- *Vertex* 0 is *adjacent* to *vertices* 1 and 2 because both {0, 1} and {0, 2} are *edges* in the graph.

- *Adjacency* describes *vertices*.
- You may think of *adjacent* as being a path of length 1.

Terminology

Incident

- Describes the relationship between vertices and *edges*. The edge {*u*, *v*} is *incident on vertices* *u* and *v* (i.e. it has *u* and *v* as endpoints).

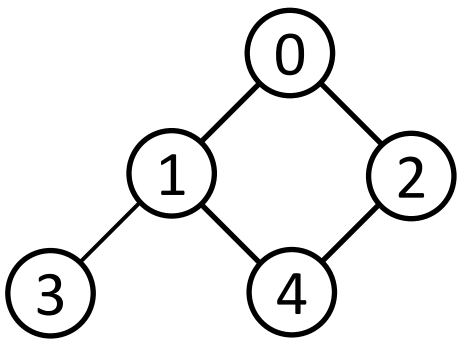


The *edges* {0, 1} and {0, 2} are both *incident on* the *vertex* 0.

Terminology

Degree of a Vertex

- The *degree* of a *vertex* is the number of *edges* incident on it.



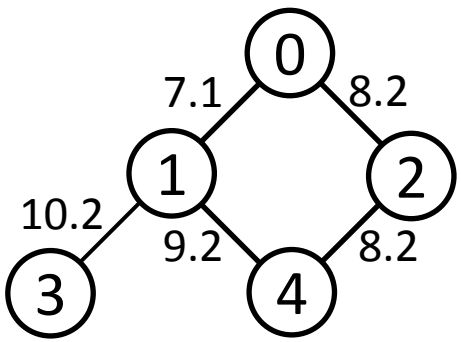
- *Vertex* 3 has *degree* 1.
  It is incident on {1, 3}.
- *Vertex* 0 has *degree* 2.
  It is incident on {0, 1} and {0, 2}.
- *Vertex* 1 has *degree* 3.
  It is incident on {0, 1}, {1, 3} and {1, 4}

- *Degree* is a property of *vertices*.

Terminology

Weighted Graph

- A *weighted graph* is a graph where each *edge* is assigned a numerical value, i.e. a *weight* (typically representing capacity or cost).



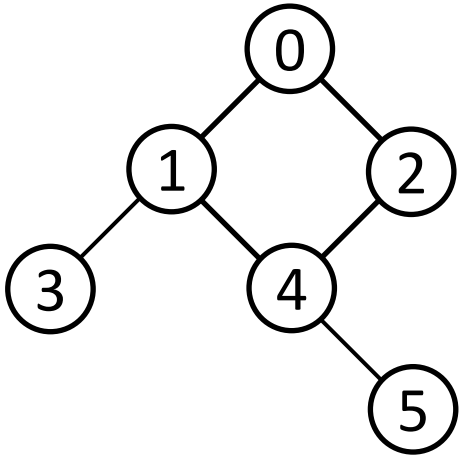
- The *weight* of the *edge* {0, 1} is 7.1
- The *weight* of the *edge* {0, 2} is 8.2
- The *weight* of the *edge* {1, 4} is 9.2
- The *weight* of the *edge* {2, 4} is 8.2
- The *weight* of the *edge* {1, 3} is 10.2

- *Weight* is a property of *edges*.

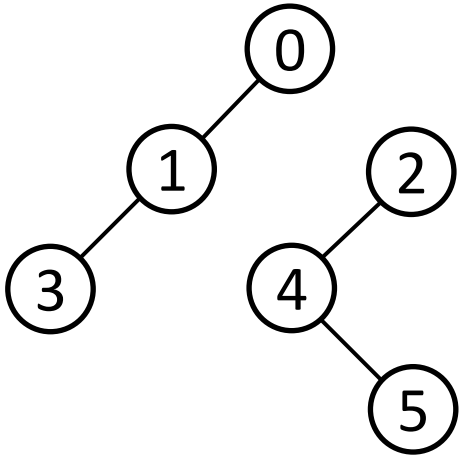
Terminology

Connected Graph

- A graph is *connected* if there is a path between any two pair of *vertices* in the graph.



A *Connected* Graph



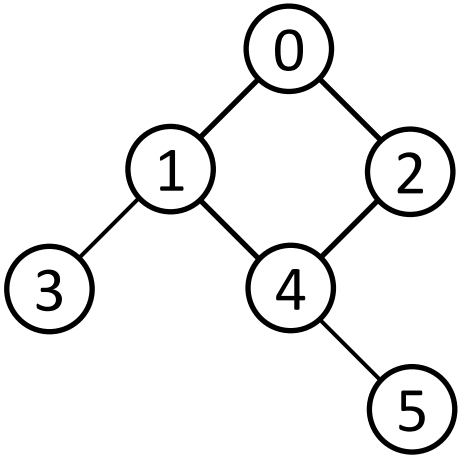
A Component Graph with 2 components

- We will be investigating *connected* graphs.

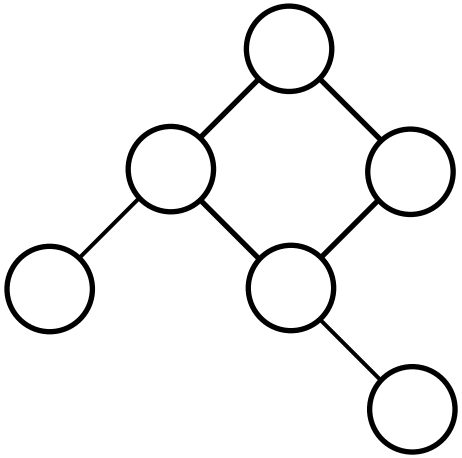
Terminology

Labelled Graph

- In a *labelled* graph, each *vertex* is assigned a unique name.



A *Labelled* Graph



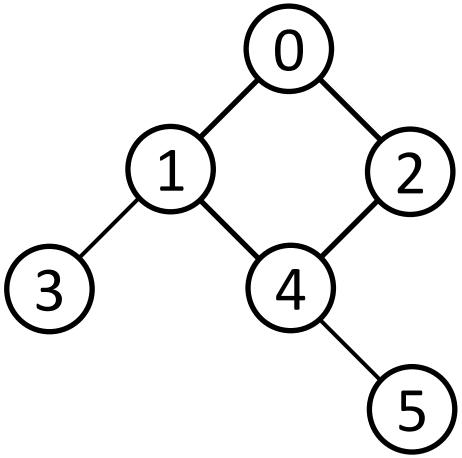
An Unlabelled Graph

- We will be investigating *labelled* graphs.

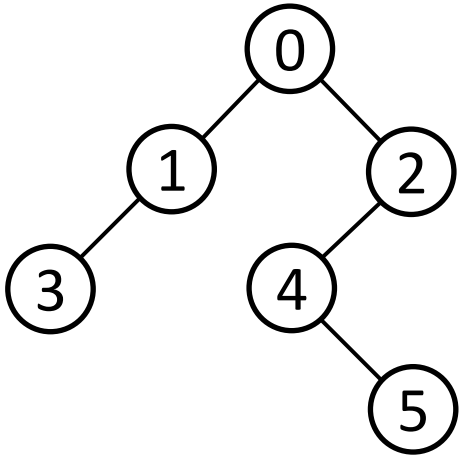
Terminology

Cyclic Graph

- A *cyclic* graph is a graph that contains a cycle, i.e. a path from a *vertex* back to itself.



A *Cyclic* Graph



An Acyclic Graph

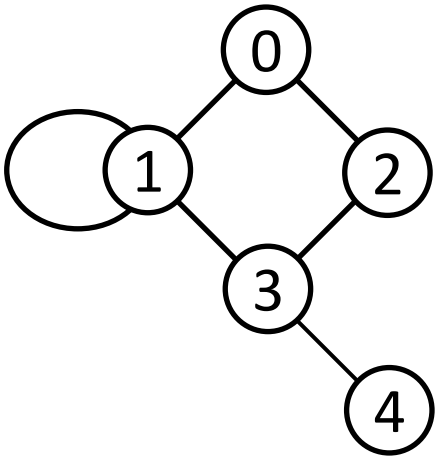
- We will be investigating *cyclic* graphs.

Terminology

Self Loops

- A *self-loop* is a an *edge* whose start and end points are the same *vertex*.

- There is a *self-loop* at *vertex* 1.



We will be investigating graphs *without self-loops*.

Graphs

Key Definitions

We will be talking a lot about *vertices*, *edges* and *adjacency* on *undirected graphs* (with *cycles* but *without self-loops*).

Reference

- Appendix B.4 Graphs of the textbook *Introduction to Algorithms* by Cormen, Thomas H. *et al.*
- 3rd edition on 3 hour loan at the Davis Library Circulation Desk, QA76.6 .C662 2009
- 2nd edition is available electronically
- the graph chapter of any text on computer algorithms

Topic 17: Notes on the Final Exam

- short answer, similar to midterm
- most important:
  - know all the algorithms
  - understand how to analyze their performance
- all material (first and second half) covered roughly equally
- there will be extra office hours before the final
- stay tuned to Piazza for an official Final Exam post

Thank-you for your attention in class and *Good Luck!*