INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

Dharwad

ज्ञानेन विकासः

Introduction to Programming and Data Structures

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur

HEIDELBERG LAUREATE FORUM

KING'S College LONDON

Indian Institute of Technology Indore

भारतीय प्रौद्योगिकी संस्थान इंदौर

PDPM

Indian Institute of Information Technology, Design and Manufacturing, Jabalpur

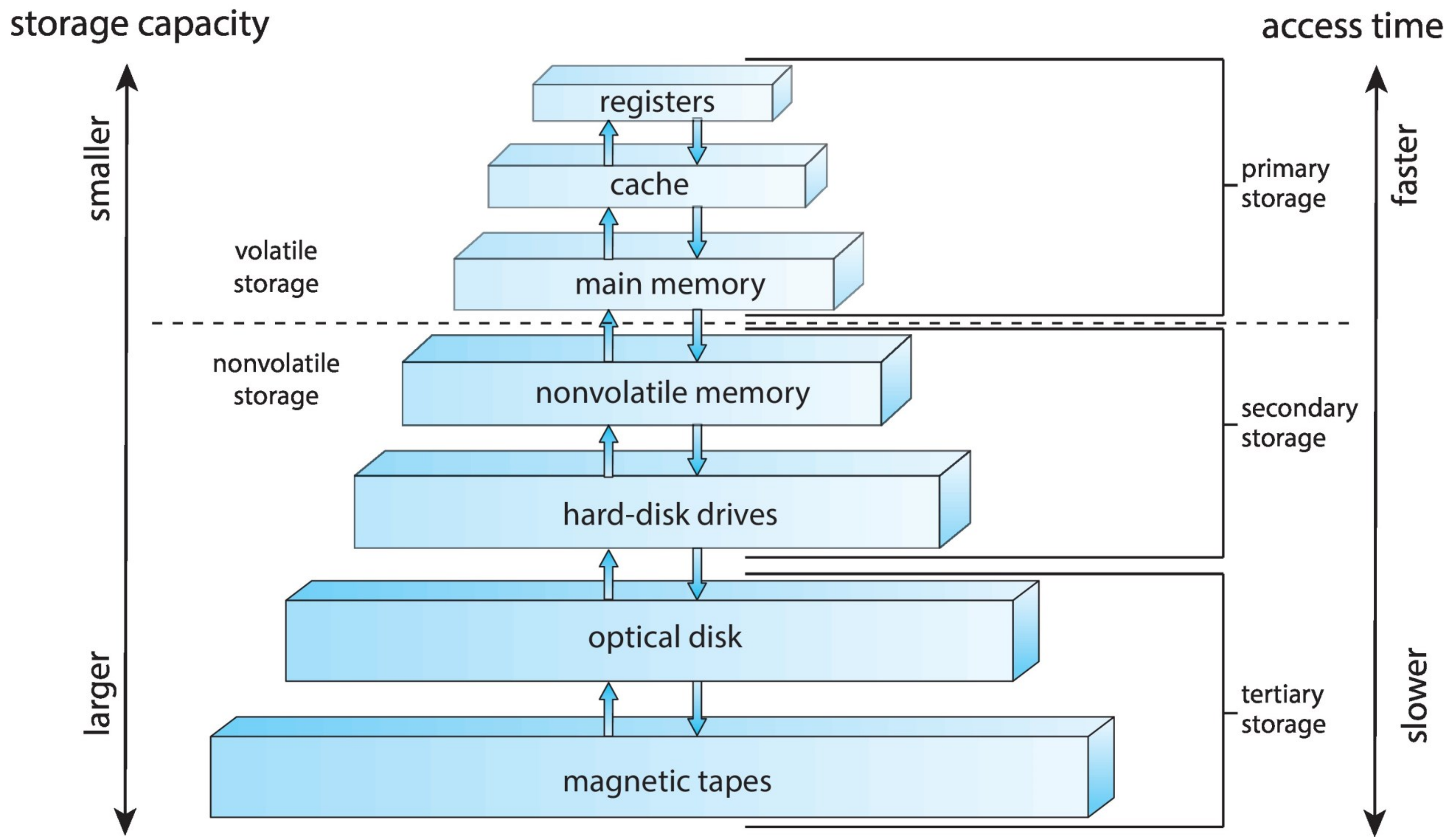
The Alan Turing Institute

Goals of the Course

- Become familiar with some of the fundamental data structures in computer science
- Improve ability to solve problems abstractly
  - data structures are the building blocks
- Improve ability to analyze your algorithms
  - prove correctness
  - gauge (and improve) time complexity
- Skill with the C, Java, and Data structures

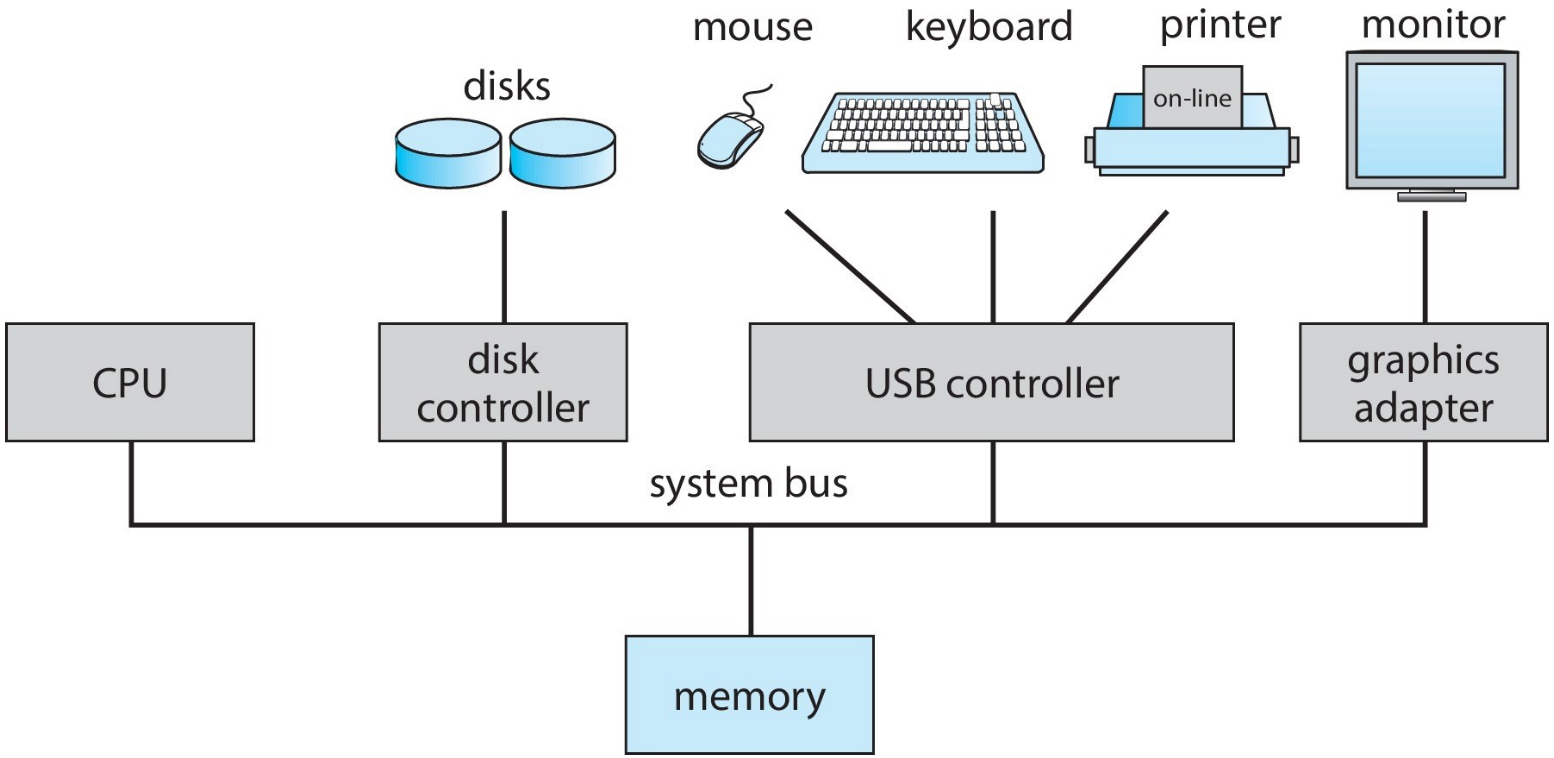
Basic Program Execution on CPU and Memory

Data Storage Device Hierarchy



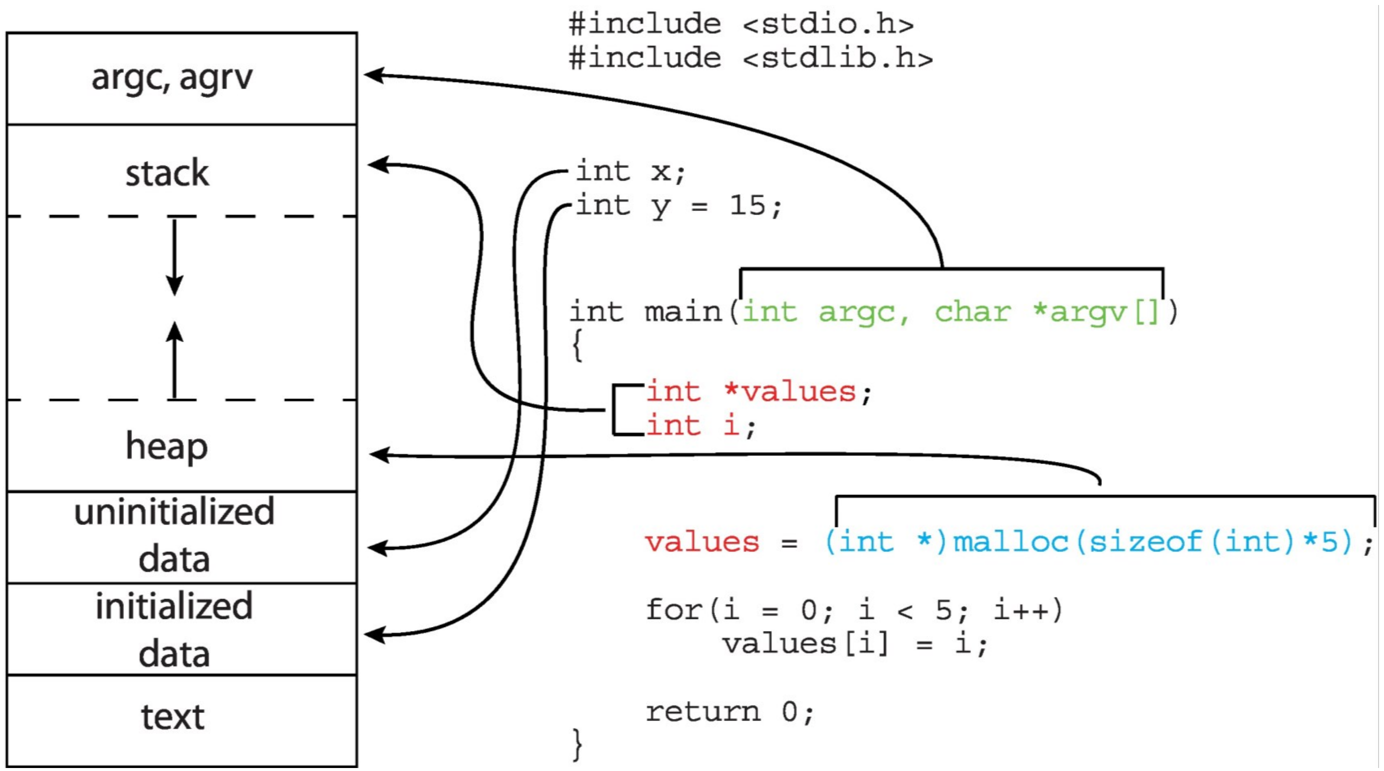
Computer System Organization

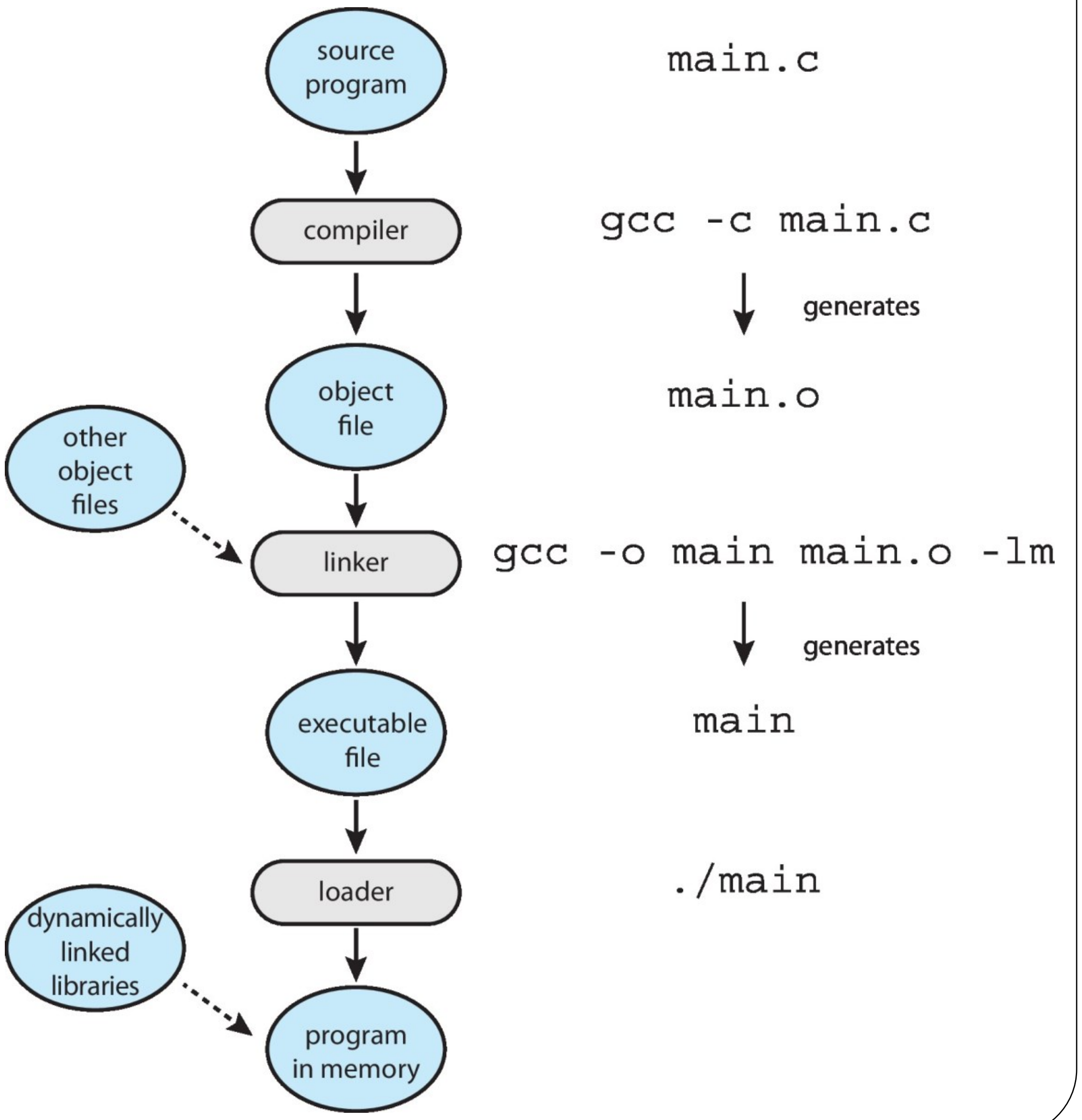
- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Program to Process

- When you run an exe file, the OS creates a process = a running program

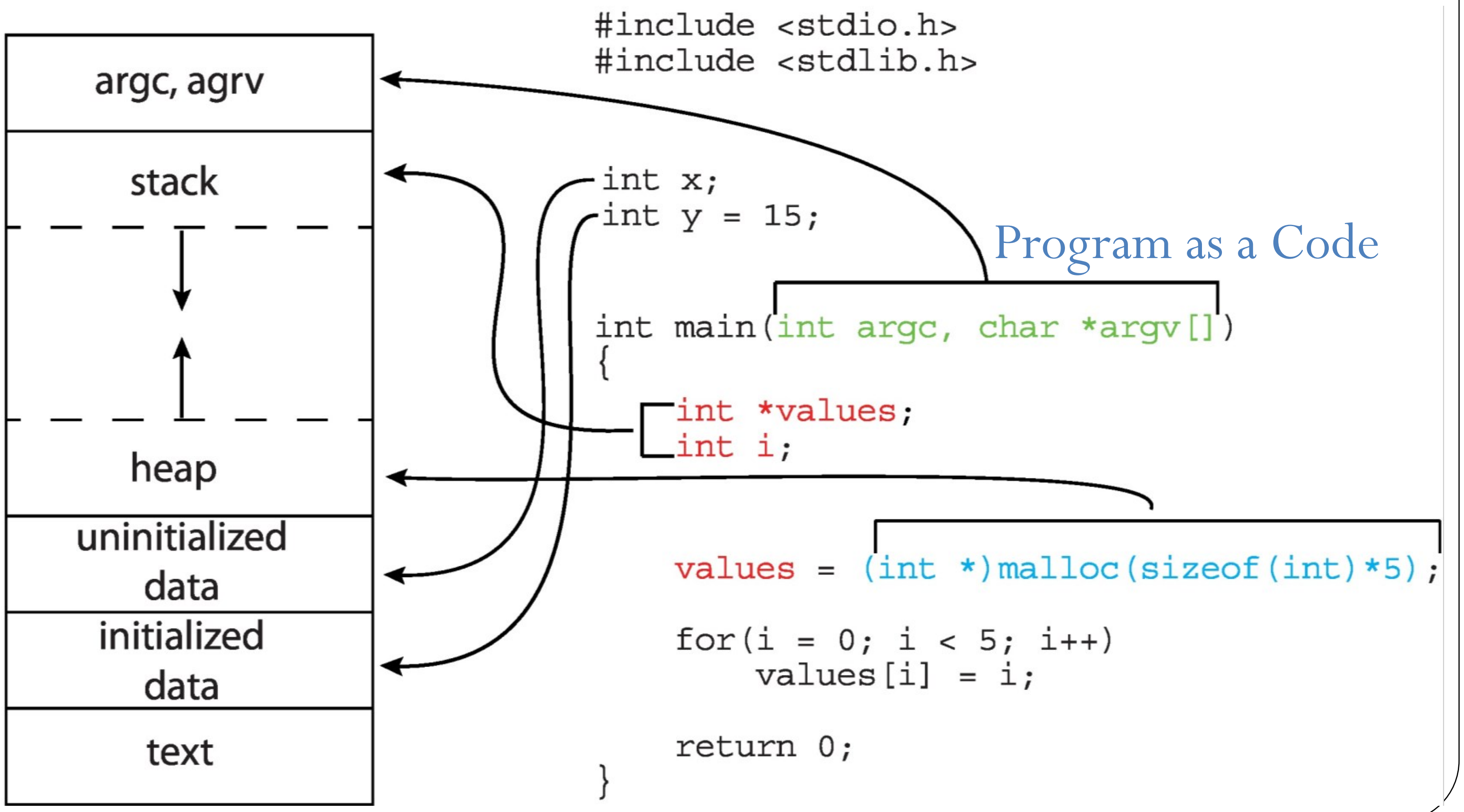




Motivation: Program Code to Memory

- **Abstraction** of complex usage Program as a Memory (RAM or Cache).
- Conversion of **High level language to Low level language**
- Static/global variables are allocated in the executable
  - Local variables of a function on Stack
  - Dynamic allocation with malloc on the heap

Process as a Memory



Program to Process

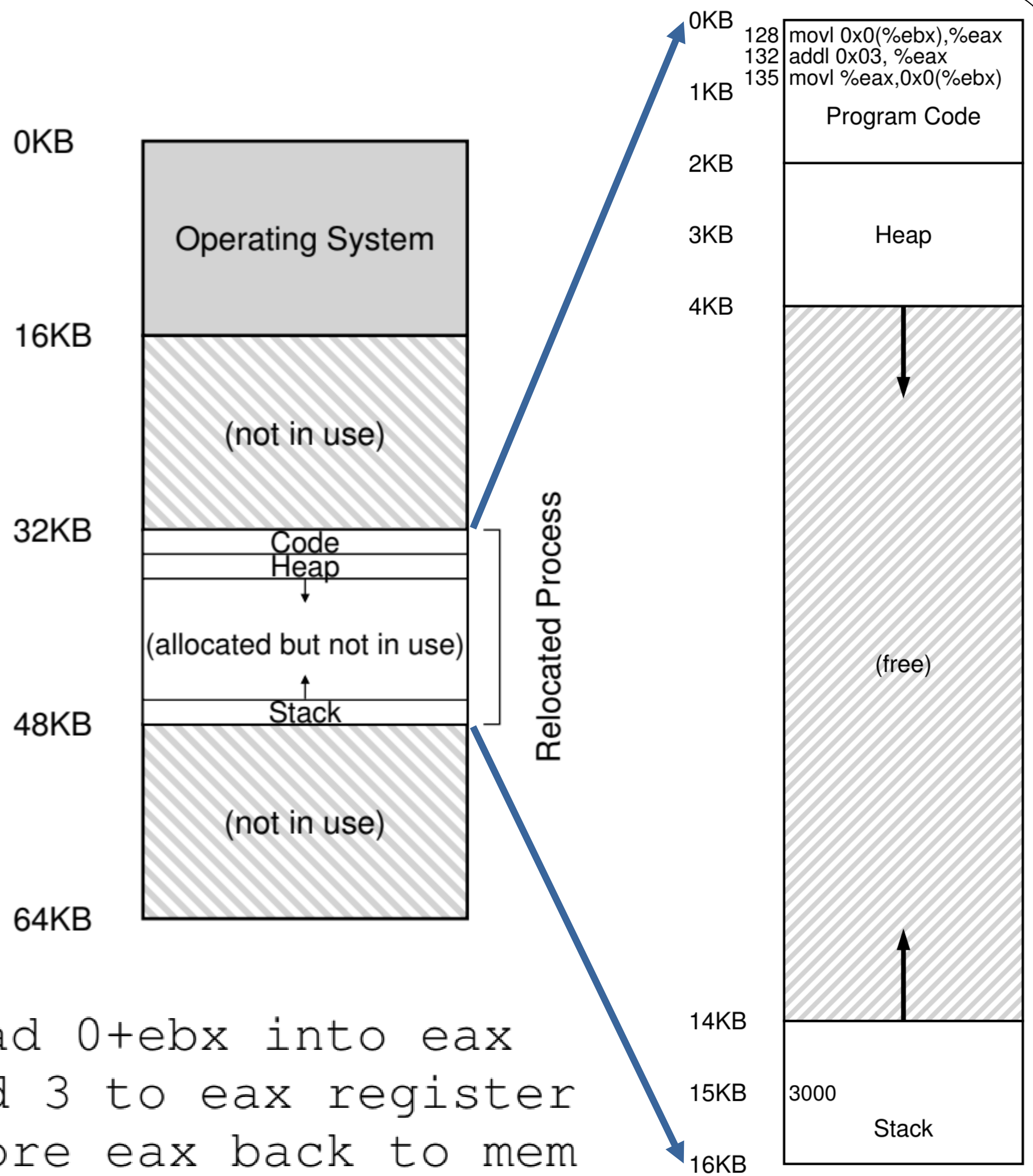
- Virtual address space is setup by OS during process creation

Simplified OS: places entire memory image in one chunk

```
void func() {
    int x = 3000;
    x = x + 3;
    ...
```

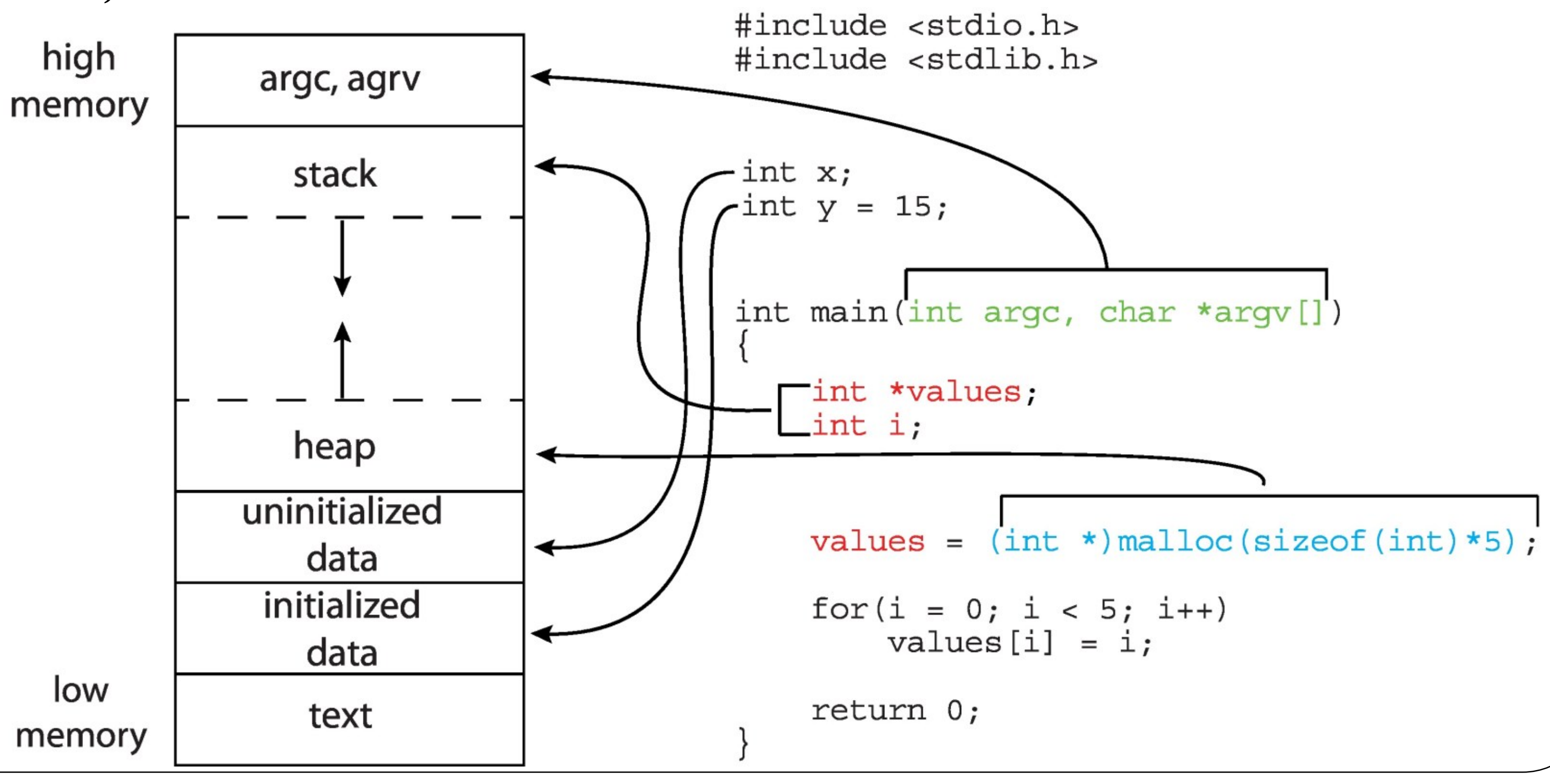
Compiler

```
128: movl 0x0(%ebx), %eax      ;load 0+ebx into eax
132: addl $0x03, %eax          ;add 3 to eax register
135: movl %eax, 0x0(%ebx)      ;store eax back to mem
```

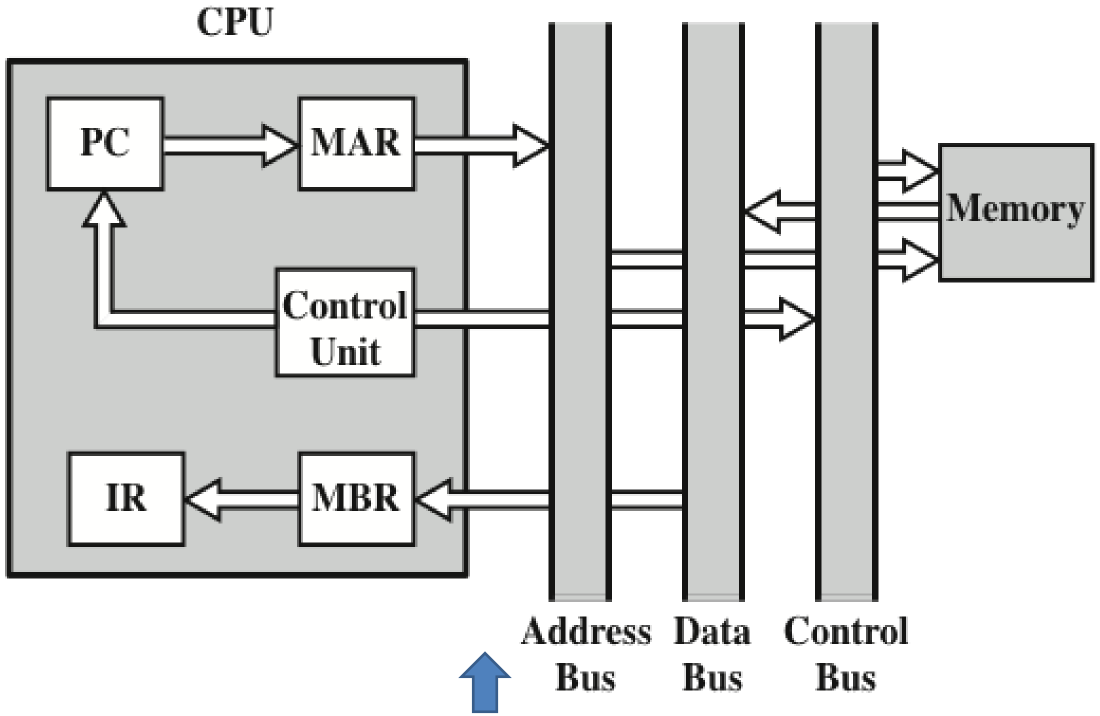


Program and Process

- A unique identifier
- Points CPU program counter to current instruction – Other registers may store operands, return values etc.
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer

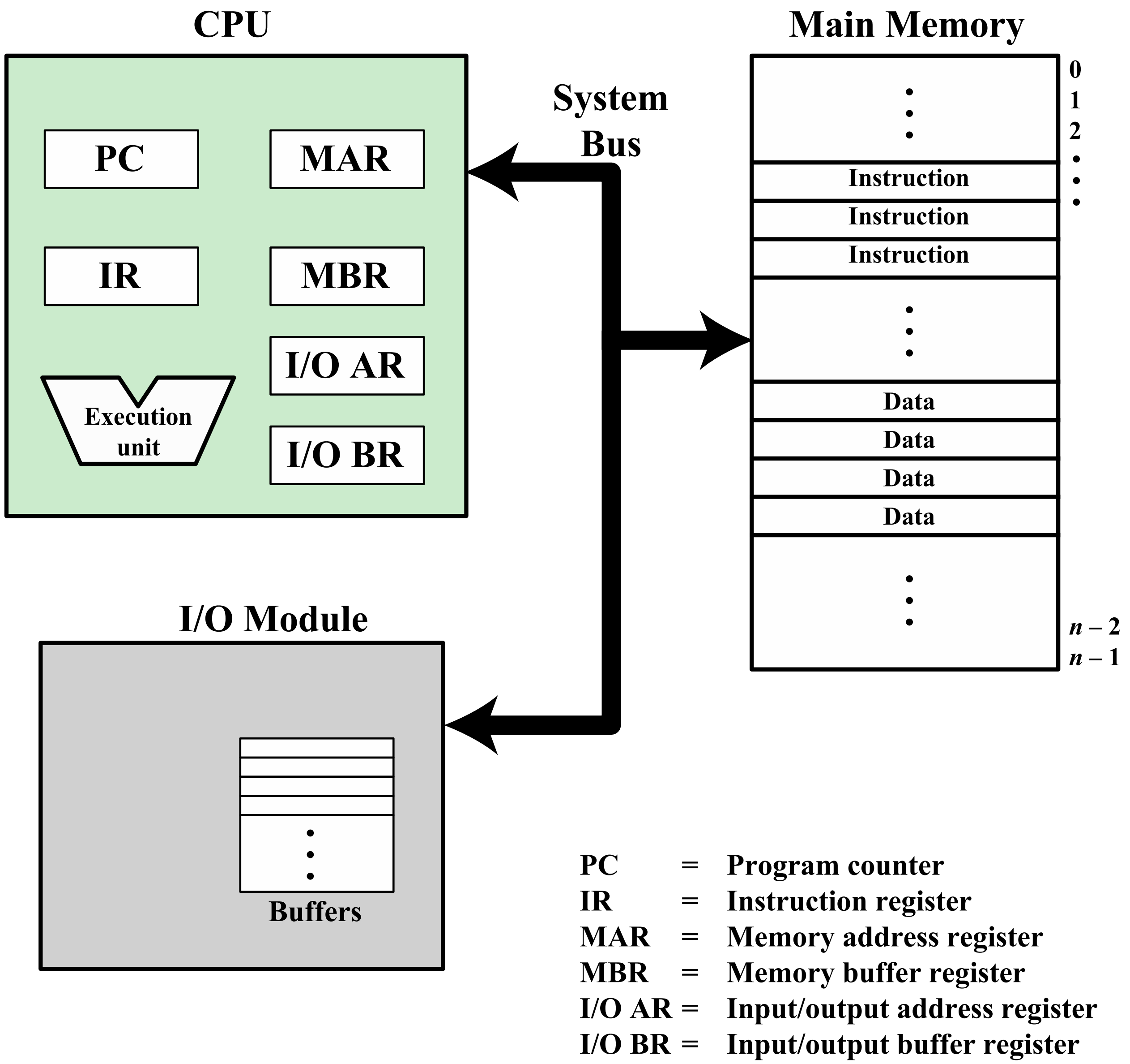


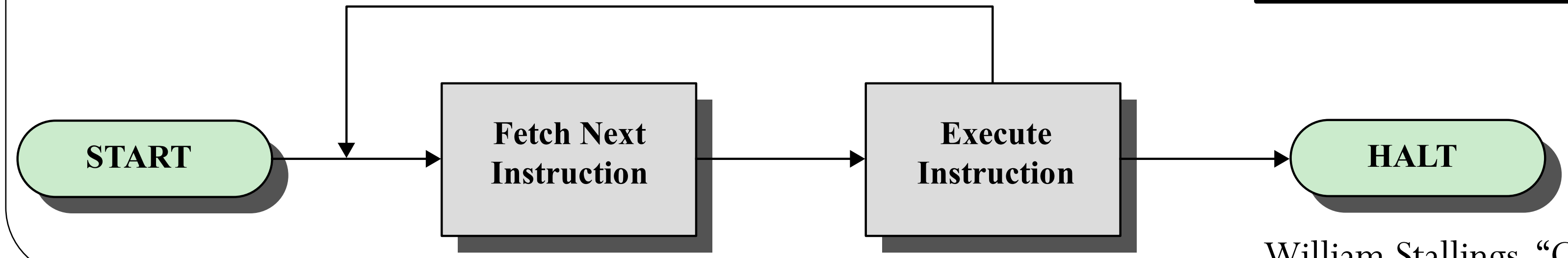
CPU and Memory



Fetch Cycle

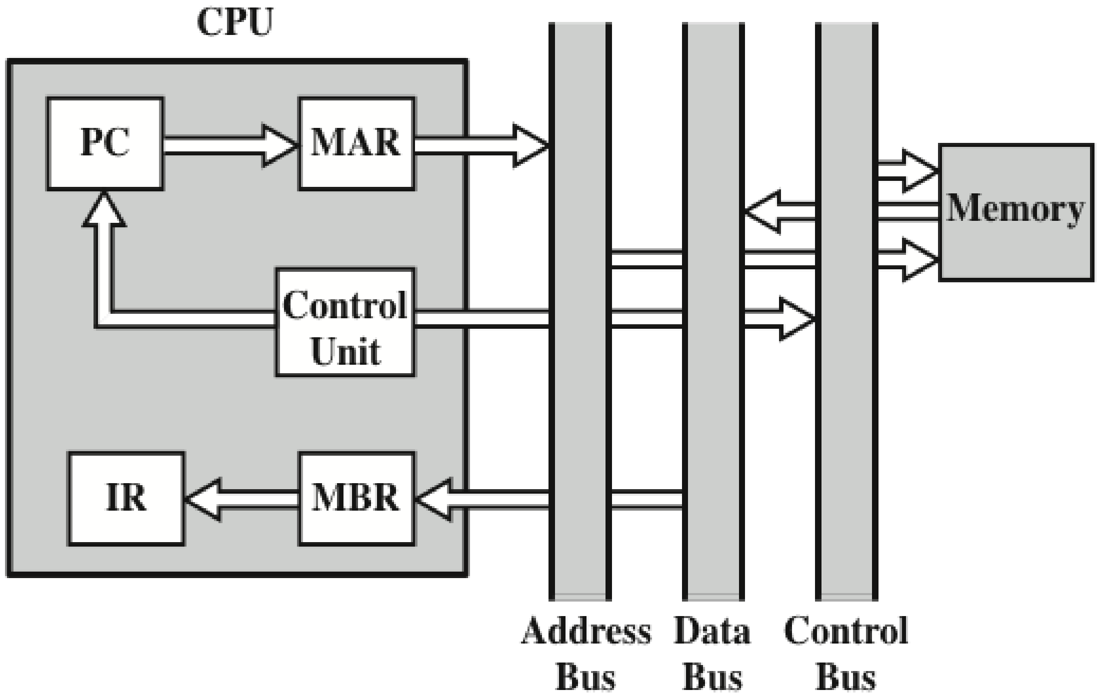
Execute Cycle





William Stallings. "Computer Organization and Architecture". 10th Edition

CPU and Memory

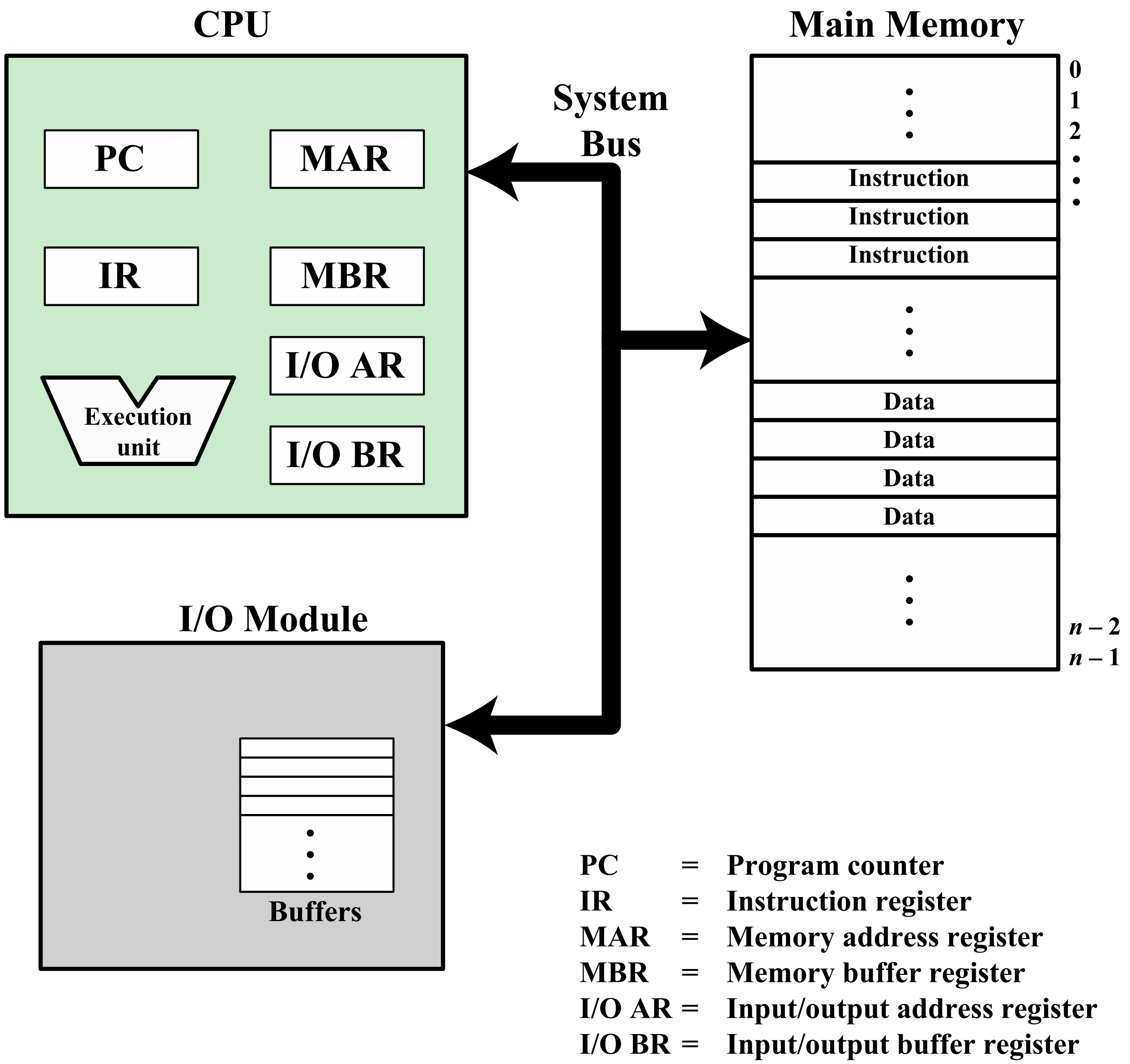


t1 : MAR ← PC

t2 : MBR ← MEMORY

PC ← (PC) + I

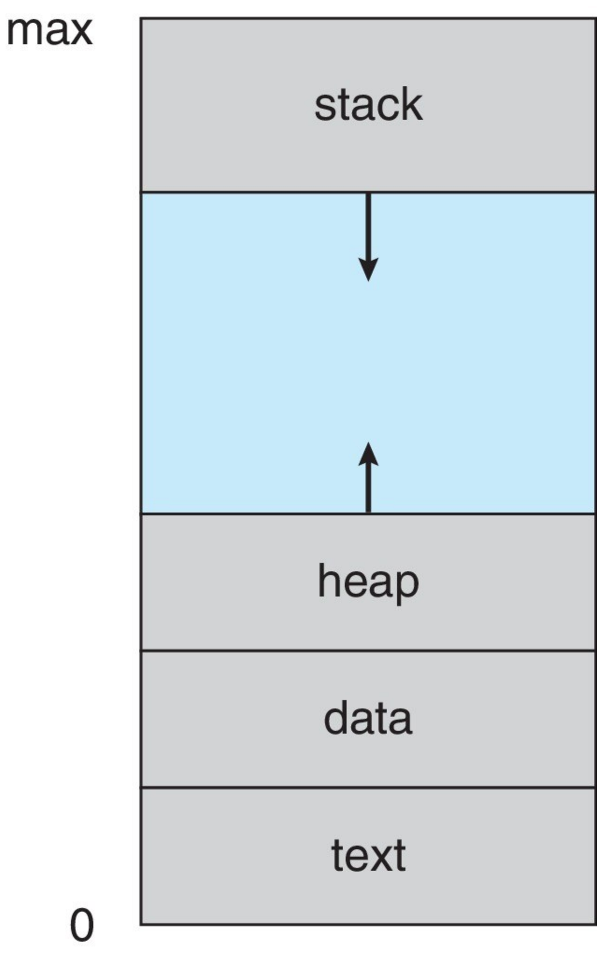
t3 : IR ← (MBR)



William Stallings. “Computer Organization and Architecture”. 10th Edition

Program and Process

- OS allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
  - Opens basic files – STD IN, OUT, ERR
  - Initializes CPU registers – PC points to first instruction
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)



Basic Data Structures

Observation

- All programs manipulate data
  - programs *process, store, display, gather*
  - data can be *information, numbers, images, sound*
- Each program must decide how to store data
- Choice influences program at every level
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

What is an Abstract Data Type?

Abstract Data Type (ADT) -

1) An opportunity for an acronym

2) Mathematical description of an object and the set of operations on the object

Data Structures : Algorithms

- Algorithm
  - A high level language independent description of a step-by-step process for solving a problem
- Data Structure
  - A set of algorithms which implement an ADT

Why so many data structures?

Ideal data structure:

fast, elegant, memory efficient

Generates tensions:

- time *vs.* space
- performance *vs.* elegance
- generality *vs.* simplicity
- one operation's performance *vs.* another's

Dictionary ADT

- list
- binary search tree
- AVL tree
- Splay tree
- Red-Black tree
- hash table

Code Implementation

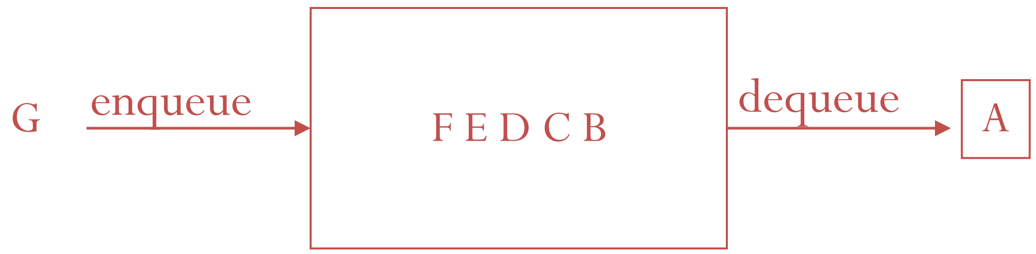
- Theoretically
  - abstract base class describes ADT
  - inherited implementations implement data structures
  - can change data structures transparently (to client code)
- Practice
  - different implementations sometimes suggest different interfaces (generality *vs*. simplicity)
  - performance of a data structure may influence form of client code (time *vs*. space, one operation *vs*. another)

ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - develop a data structure for the ADT
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

Queue ADT

- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty



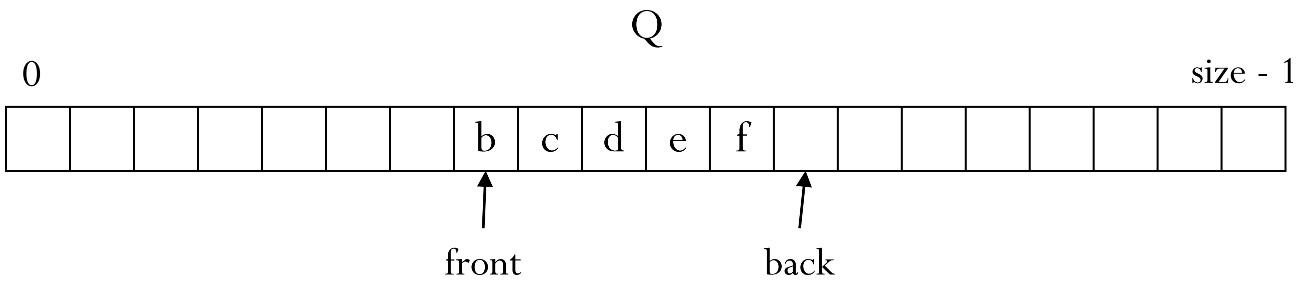
- Queue property: if x is enQed before y is enQed, then x will be deQed before y is deQed

  FIFO: First In First Out

Applications of the Q

- Hold jobs for a printer
- Store packets on network routers
- Hold memory “freelists”
- Make waitlists fair
- Breadth first search

Circular Array Q Data Structure




```
void enqueue(Object x) {
  Q[back] = x
  back = (back + 1) % size
}
Object dequeue() {
  x = Q[front]
  front = (front + 1) % size
  return x
}
```

When is the Q empty?

Are there error situations this code will not catch?

What are some limitations of this structure?

This is *pseudocode*. Do not correct my semicolons.

Q Example

enqueue R

enqueue O

dequeue

enqueue T

enqueue A

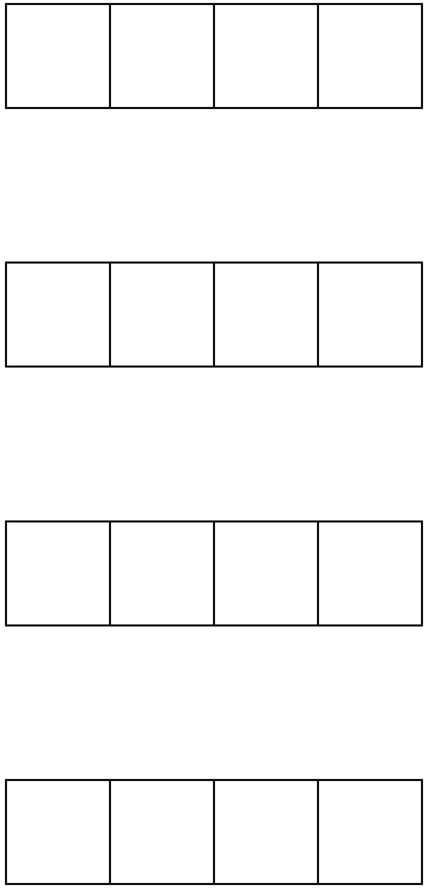
enqueue T

dequeue

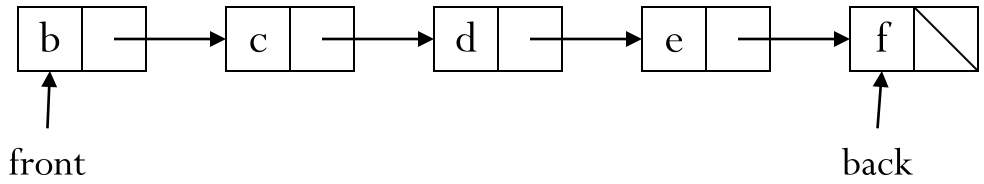
dequeue

enqueue E

dequeue



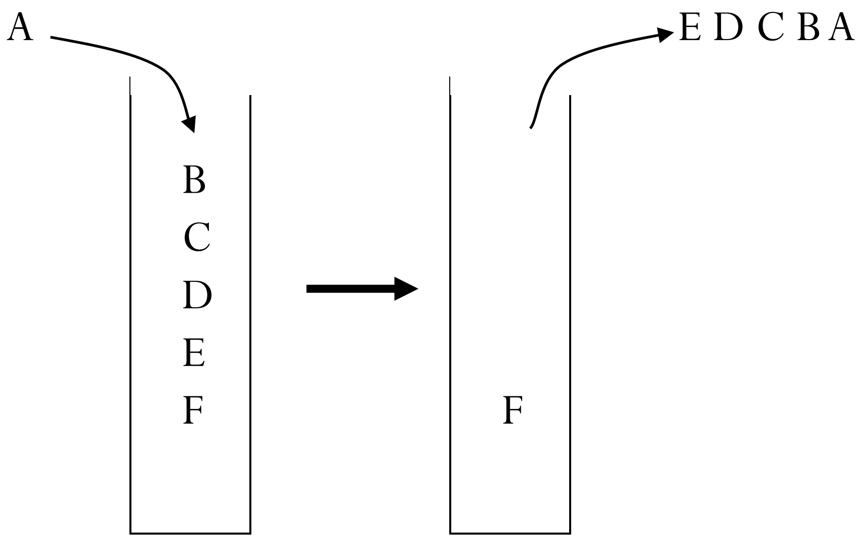
Linked List Q Data Structure




```
void enqueue(Object x) {
   if (is_empty())
       front = back = new Node(x)
   else
       back->next = new Node(x)
       back = back->next
}
```

```
Object dequeue() {
   assert(!is_empty)
   return_data = front->data
   temp = front
   front = front->next
   delete temp
   return temp->data
}
```

LIFO Stack ADT



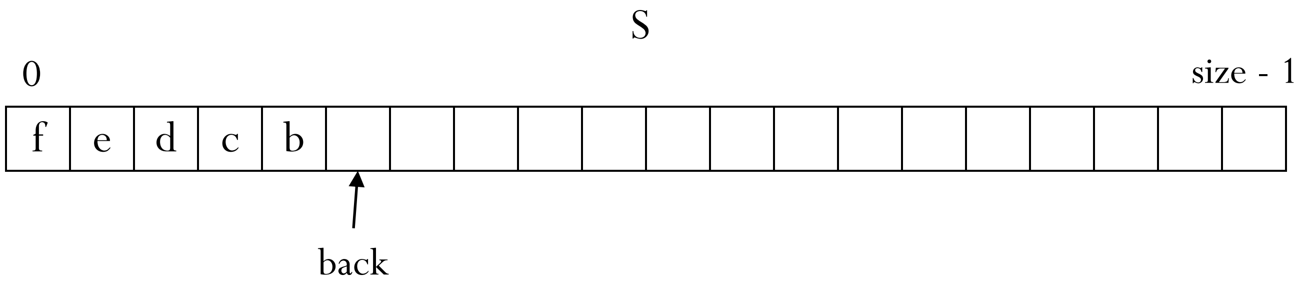
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty
- Stack property: if x is on the stack before y is pushed, then x will be popped after y is popped

  LIFO: Last In First Out

Stacks in Practice

- Function call stack
- Removing recursion
- Balancing symbols (parentheses)
- Evaluating Reverse Polish Notation
- Depth first search

Array Stack Data Structure

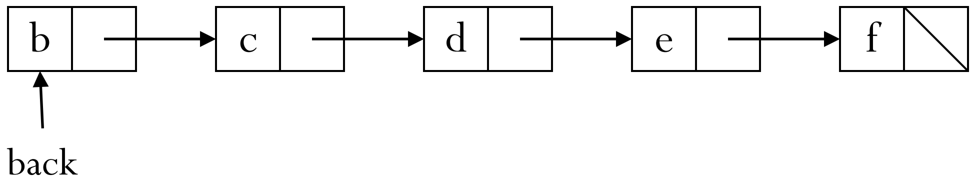



```
void push(Object x) {
  assert(!is_full())
  S[back] = x
  back++
}
Object top() {
  assert(!is_empty())
  return S[back - 1]
}
```

```
Object pop() {
  back--
  return S[back]
}

bool is_full() {
  return back == size
}
```

Linked List Stack Data Structure




```
void push(Object x) {
   temp = back
   back = new Node(x)
   back->next = temp
}
Object top() {
   assert(!is_empty())
   return back->data
}
```

```
Object pop() {
   assert(!is_empty())
   return_data = back->data
   temp = back
   back = back->next
   return return_data
}
```

Data structures you should already know

- Arrays
- Linked lists
- Trees
- Queues
- Stacks

ขอบคุณ

Thai

Grazie

Italian

תודה רבה

Hebrew

धन्यवादः

Sanskrit

ಧನ್ಯವಾದಗಳು

Kannada

Ευχαριστώ

Greek

Thank You

English

Gracias

Spanish

Спасибо

Russian

Obrigado

Portuguese

شكراً

Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci

French

多謝

Traditional Chinese

धन्यवाद

Hindi

Danke

German

多谢

Simplified Chinese

நன்றி

Tamil

Tamil

ありがとうございました

Japanese

감사합니다

Korean