Chapter 2 - Background

Introduction to Programming Languages

First Edition, 2013

Author: Arvind Bansal
Kent State University
Kent, OH 44242

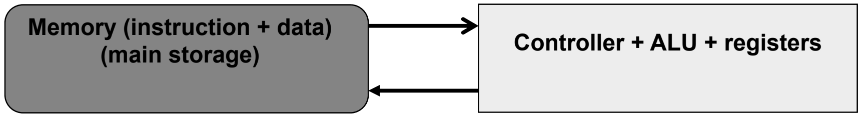
Topics covered

- von-Neuman machine
- Discrete structure concepts
- Data structure concepts
- Abstract concept in computation
- Operating system concepts
- Summary

---

von-Neuman Machine

- **von-Neuman machine**
  - Memory locations, central processing unit, registers, program counters and data bus
  - Instructions and data are stored in memory
  - Program counter moves to next instruction
  - Four stage cycle: fetch instruction → decode → compute → store
- **Instructions have opcode and operands**
  - Number of operands are called address mechanisms
  - add R1, R2, R3 % 3 – address mechanism
- Categories of instructions
  - **load, store, move, arithmetic, comparison, logical, conditional and unconditional jumps, storing the computation status**



Memory (instruction + data) (main storage)

Controller + ALU + registers

Addressing Mechanisms

- **Memory holds instruction and data**
  - first part of an instruction is called opcode; remaining part is data
  - data can be stored in processor register or memory
  - depending on decoding more memory locations may be fetched
- **Machine can be**
  - zero-address machine: uses a stack and stack based operations
  - one–address machine: uses a default register called accumulator
  - two address machine: result is stored in the second address
  - three address machine: src1 + src2 → destination
- **Schematic of different instructions:**
  - zero-address machine: push; pop; load; store etc.
  - one address machine: load memory % Acc → memory
  - two address machine: add R1, R2 % R1 + R2 → R2
  - Three address machine: add R1, R2, R3 % R1 + R2 → R3

Instruction Types

Instruction Type	Type of operations
Three-address instruction <opcode> <src$_1$>, <src$_2$>, <dst>	An instruction in 3-address machine. Instruction could be any arithmetic or logical dyadic operation
Two-address instruction <opcode> <src$_1$>, <src$_2$>	An instruction in 2-address machine. Instruction could be any arithmetic or logical dyadic operation. The destination is same as the second argument.
One-address instruction <opcode> <src>	Instruction could be any arithmetic or logical dyadic operation. One of the registers by default is accumulator that acts as destination
Zero-address instruction <instr-name>	Load, add, subtract, multiply, load, store etc. Uses a stack based evaluation. Argument for operations are picked up from top of the stack

Efficiency of Addressing

- A address machine instructions can be converted to one or more lower level machine instruction
- Instructions in 3-addr < 2-addr< 1-addr < 0-addr for the same task
- time taken to execute same task is more in low level instructions
- **Example of Conversion**

Two address machine	Zero Address Machine
load A, R0 % R0 ← content-of(A) integer_add B, R0 % R0 ← R0 + % content-of(B) store R0, X % store R0 into X	load_literal 3 % load the address of X load_literal 1 % load the address of A load % load the value of A on top load_literal 2 % load the address of B load % load the value of B on top integer_add % pop and add 2 numbers store % store the top

Sets and Bags

Set = {a, b, c}; bag = {a, b, a, c, b};

- **Set operations are used in**
  - Type theory to form structured types like array, struct, tree etc.
  - Set based programming
- **Set and bag**
  - A set has unique entities; a bag may have more than one element having the same value
- **Unordered set vs. ordered set**
  - Shuffling the elements in a set does not change the unordered set
  - Shuffling the elements changes an ordered set because elements are associated with positions
- **Similar difference between ordered and unordered bag**

Set Operations I

Sets: X = {a, b, c}; Y = {1, 2, 3}; Z = {m, b, x}

- **Union: X U Z = {a, m, b, c, x}**
- **Intersection: X ∩ Z = {b}**
- **Difference: X – Z = {a, c}**
- **Power set**
  - set of all subsets of the original set
  - Example: powerset(X) has 8 elements: {{ }, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}
  - Two possibilities for each element: absent or present
  - Total possibilities = $2^{|N|}$ where |N| = number of elements in the original set

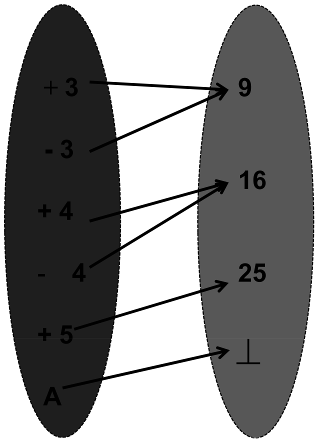
Set Operations II

Sets: A = {a, b, c}; B = {1, 2, 3};

- **Cartesian product**
  - An exhaustive set of n-tuples where n is the number of sets involved in Cartesian product
  - Ith element of n-tuple comes from the Ith set
  - **Example**
    - A X B = {(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3), (c, 1), (c, 2), (c, 3)}
- **Disjoint Union**
  - Union of two disjoint sets
  - elements of the same sets are colored (distinct tags)
  - **Example**
    - A ⊎ B = {(true, a), (true, b), (true, c), (false, 1), (false, 2), (false, 3)} where true and false are two colors
  - : A ⊎ B = {true} X A ⋃ {false} X B

Set Operations III

- **Mapping**
  - mapping one or more elements of a set to an element of 2nd set
  - one-to-one onto, one-to-one into, many-to-one onto, many-to-one into
  - finite domain → finite mapping
- **Lifted Domain**
  - uses bottom symbol ⊥
  - undefined mappings map an element to bottom symbol



Boolean Logic

- **There are basic axioms that can be true or false**
- **New axioms can be derived from basic axioms and logical operators**
- **Logical operators can be negation, logical-and (∧) , logical-or (∨), implication (→) etc.**

A	**B**	**¬(A)**	**A ∧ B**	**A ∨ B**	**A → B**
true	false	false	false	true	false
false	false	true	false	false	true
true	true	false	true	true	true
false	true	true	false	true	true

Equivalence of Operations

Operations	Equivalence		
Negation	$\neg(\neg P_1)$	≡	P_1
Associativity	$P_1 \wedge (P_2 \wedge P_3)$	≡	$(P_1 \wedge P_2) \wedge P_3$
	$P_1 \vee (P_2 \vee P_3)$	≡	$(P_1 \vee P_2) \vee P_3$
Commutativity	$P_1 \wedge P_2$	≡	$P_2 \wedge P_1$
	$P_1 \vee P_2$	≡	$P_2 \vee P_1$
Distributivity	$P_1 \wedge (P_2 \vee P_3)$	≡	$(P_1 \wedge P_2) \vee (P_1 \wedge P_3)$
	$P_1 \vee (P_2 \wedge P_3)$	≡	$(P_1 \vee P_2) \wedge (P_1 \vee P_3)$
De Morgan's rule	$\neg(P_1 \wedge P_2)$	≡	$(\neg P_1) \vee (\neg P_2)$
	$\neg(P_1 \vee P_2)$	≡	$(\neg P_1) \wedge (\neg P_2)$

First Order Predicate Calculus

- **Propositional calculus + quantification**
- **Two types of quantification**
  - Universal quantification (∀) associates a property with all the elements of a set.
  - Existential quantification (∃) finds an element satisfying a property.
- **Example**
  - ∀X (man(X) → likes_to_live_longer(X)).
  - ∀X ∀Y (sibling(X, Y) ← ∃Z (parent(X, Z), parent(Y, Z), not (X == Y)

Operations	Equivalences		
Commutativity	∀x ∀y P(x, y)	≡	∀y ∀x P(x, y)
	∃x ∃y P(x, y)	≡	∃y ∃x P(x, y)
Duality	∀x P(x)	≡	-∃x (¬ P(x))
	∃x P(x)	≡	¬∀x (¬ P(x))

Relations and Functions

- **Relation**
  - R ⊆ A X B such that (x ∈ A, y ∈ B) ∈ R
  - Relationship can also be denoted as xRy or R(x, y).
  - Relation can be reflexive: xRx; transitive: xRy and yRz → xRz or symmetric: xRy → yRx
  - Inverse of symmetric is antisymmetric; xRy is not same as yRx
- **Function**
  - A single valued mapping from an element in a domain to another element in the codomain
  - The set of images of the domain elements is called range
  - Identity function maps the element to itself
  - Onto function: every element in codomain is an image
  - One-to-one: there is a unique image for every domain element
  - Bijective function; one-to-one and onto

Recursion

- **A definition uses itself to define with different argument**
  - At least one base case and at least one recursive definition
  - Progressively unfolds and moves towards the base case
  - Previous invocations are suspended until next recursive invocation returns value
  - Number of invocations decided by the input value
- **Example: factorial function or fibonacci function**

factorial(0) = 1. % base clause
factorial(n) = n * factorial(n – 1) % recursive definition

fibonacci(0) = 1.
fibonacci(1) = 1.
fibonacci(n) = fibonacci(n – 1) = fibonacci(n – 2).

Recursion vs. Iteration

- Implementation of recursion
  - A stack is needed to hold the execution space needed by variables for every procedure invocation
  - Stack has memory and execution overhead of calling and returning from called recursive procedures.

Iteration	Recursion
No overhead of calling and returning from called procedure	Excessive overhead of calling and returning from recursive invocation
starts from the base case, and reuses the memory locations to accumulate results	Suspends recursive calls that needs additional memory before hitting the base case
more efficient execution	Slow due to overheads

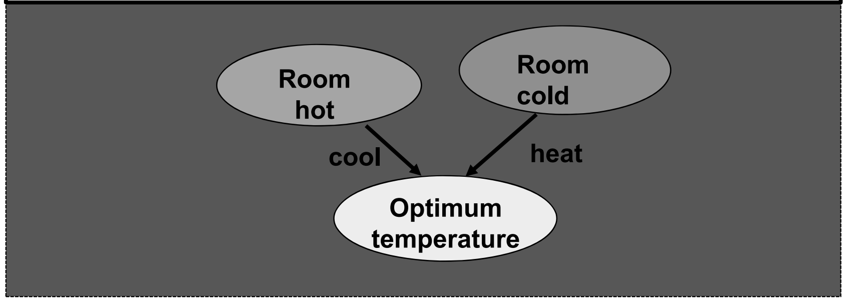
Optimizing Recursive programs

- **Tail recursion**
  - Recursive call is the last one in the definition.
  - Is equivalent to indefinite iteration.
  - Tail recursion can be transformed to equivalent indefinite iteration
- **Linear recursive programs**
  - Has only one recursive call to itself in the recursive definition
  - Can be transformed to indefinite indefinite iteration

```
Algorithm iterative_factorial
Input: input  value n;
Output: accumulator  value;
{    acc = 1;
     for (i = 1; i =< n; i++)
          acc  = i *   acc;
}
```

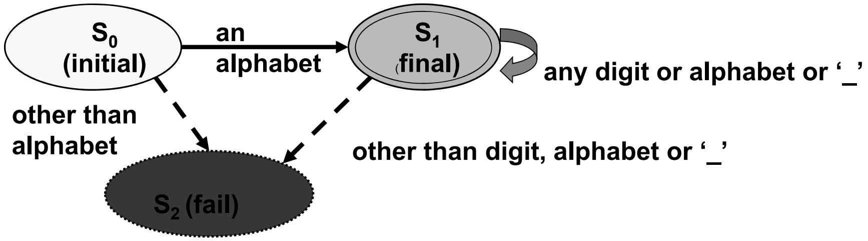
Finite State Automata (FSA)

- Abstract machine to model real world phenomenon
  - Different situations modeled as states
  - Transitions modeled by edges
  - Initial state – where the machine starts
  - Final state – where the transitions finally terminate
  - Used in accepting specific sequence patterns during compilation



FSA in Accepting Variable-names

- **Variable: alphabet followed by sequence of alphabet or numbers**
- **FSA**
  - Initial state S0; final state S1; error state: S2
  - Transition: S0 → S1 upon the first alphabet
  - Transition S1 → S1 upon alphabet / number
  - Transition S0 → S2 if any character other than alphabet
  - Transition S1 → S2 if any character other than alphabet / number



Sequence

- **Definition: a bag to model ordered collection of entities**
- **Representation: modeled within angular brackets**
  - <a, b, c>
- **Operations**
  - Finding an element by position
  - Insertion and deletion of elements by index
  - First, second, last elements of a sequence
  - Deletion and substitution of a subsequence by content
  - Joining two sequences
  - Finding out predecessor and successor of an entity in a sequence
- **Application**
  - multiple data types can be modeled as sequence such as stacks, queues, files, strings etc.

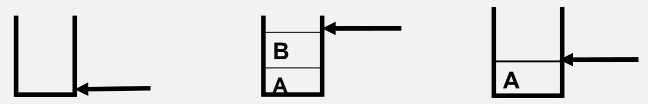
Operations on Sequence

Operation	Output	Explanation
first($<x_1 \ldots x_n>$)	x_1	the first element of a sequence
last($<x_1 \ldots x_n>$)	x_n	the last element of a sequence
rest($<x_1 \ldots x_n>$)	$<x_{2,\ldots,}x_n>$	the rest of the sequence
butlast($<x_1 \ldots x_n>$)	$<x_{1,\ldots,}x_{n-1}>$	the subsequence except the last element
Nth(i, ($<x_1 \ldots x_n>$)	X_i	ith element of a sequence
cons(a, $<x_1 \ldots x_n>$)	$<a, x_{1,\ldots,}x_n>$	constructs sequence by adding a in the beginning of the old sequence
insert(i, a, $<x_1 \ldots x_n>$)	$<x_{1,\ldots}x_{i-1,,}a_{,}x_{i+1,\ldots}\ x_n>$	'a' is inserted as the ith element
append($<x_1 \ldots x_n>$, $<y_1 \ldots y_m>$)	$(<x_1 \ldots x_n, y_1 \ldots y_m>$	Concatenate sequences in order
subseq($<x_1 \ldots x_n>$,i, m)	$<x_i,\ldots,x_{i+m}>$	A subsequence starting from start location I of length m
is_subseq($<x_1 \ldots x_n>$, $<y_1 \ldots y_m>$)	Boolean	Returns true if $<x_1 \ldots x_n>$ is included in $<y_{1,}\ldots y_m>$ otherwise returns false

---

Stack

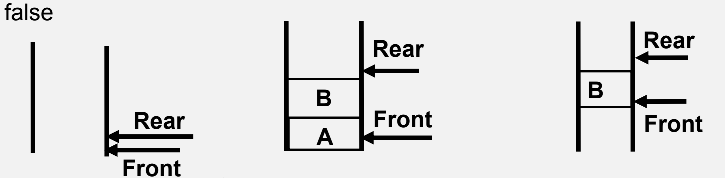
- **Definition: a sequence where data can be inserted and deleted from one end; other end is sealed.**
  - also called LIFO – Last In First Out
- **Abstract operations: push, pop, top, is_empty**
  - push takes an element and puts on the top of the stack
  - pop removes the top element of the stack
  - top just reads the top element of the stack
  - Is_empty returns true if the stack is empty else returns false



(a) empty stack **Pushing A then B** **After pop operation**

Queue

- **Definition: a sequence where data can be inserted from the front end and removed from the rear end.**
  - Also called FIFO – First In First Out
- **Abstract operations: insert, remove, first, is_empty**
  - The operation **'insert'** inserts an element at the rear pointer.
  - The operation '**remove'** removes the first element pointed by the front pointer.
  - The operation **'first'** reads the first element pointed by the front pointer.
  - The operation '**is_empty'** returns true if the queue is empty else returns false



Empty queue **After inserting A then B** **After remove**

Data and Reference

- **Memory locations hold two types of information**
  - Data and reference to memory locations
  - Pointers are addresses of memory location stored in another memory location or processor registers
- **Advantages of pointers**
  - Minimal overhead of data movement
  - Supports recursive data structures (lists, trees) and dynamic objects
  - Delaying memory allocation of variables until runtime
  - Allocating physically separated chained memory blocks for logically contiguous data structures
  - Sharing memory blocks among multiple data structures
  - Providing independence of the program from data movement
- **Disadvantages of pointers**
  - Arithmetic operations on pointers causes segment hopping error.
  - Shared blocks can not be reused until all pointers are released.

Recursive Data Structures

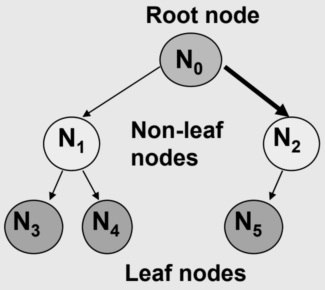
- **Definition: A data structure that uses itself in the definition**
  - One or more recursive definition and one or more base definition
- **Examples: linked list, tree, vector**
- **Linked list**

  <linked-list> ::= *<data-element>* *<linked-list>* | null
- Trees

  <binary-tree> ::= *<binary-tree>* *<data-element>* *<binary-tree>* | *void*
- **Implementation**
  - Uses pointers or references to implement.
  - Pointers are used so that memory allocation can be done at runtime as needed.

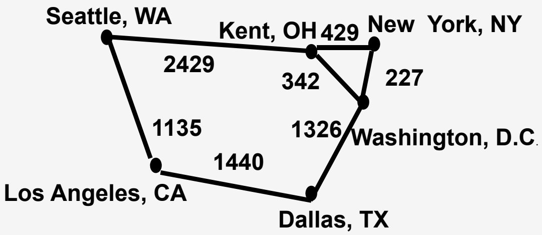
Trees

- **Tree is a recursive data structure**
  - A node connected to multiple subtrees
  - Node hold information along with pointers to the subtrees
  - Has a root node (top level node), nonleaf nodes and leaf nodes
  - Pointers connect parent node to children nodes
- **Some types of trees**
  - Binary tree: at most two branches
  - Tertiary tree: at most three branches
  - Quad-tree: four branches
  - n-ary tree: at most N branches
  - AND-OR tree
- **Binary tree**
  - Complete binary tree
  - Almost complete binary tree



Graphs

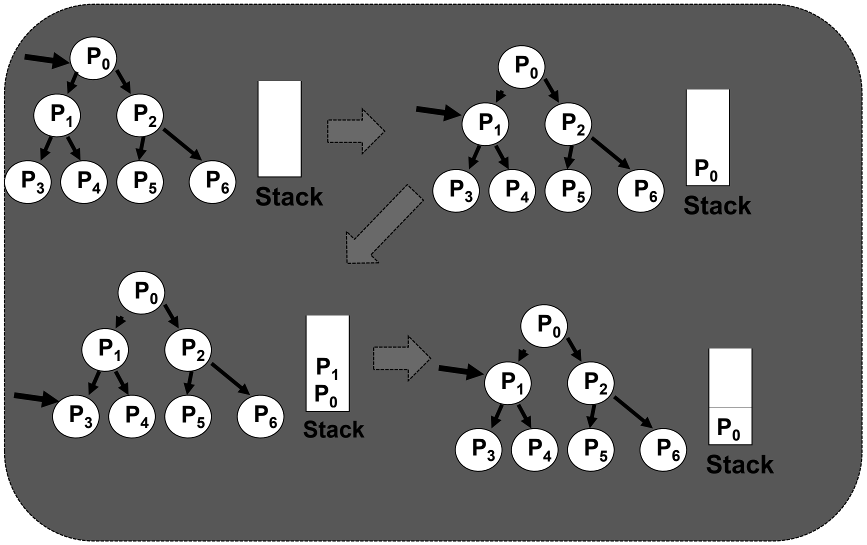
- **Definition: a set of vertices and connecting edges**
- **Trees vs. Graphs**
  - Trees have only one incoming edge from parent. Graphs may have many incoming edges
  - Graphs may have cycle that means using a non-repeating sets of edges one can come back to the vertex. Trees have no cycles
  - Trees are special case of graphs with no cycles
- **Types of graphs**
  - Weighted graphs
  - Directed graphs
  - Cyclic graphs
  - Acyclic graphs
  - Directed acyclic graphs (DAG)



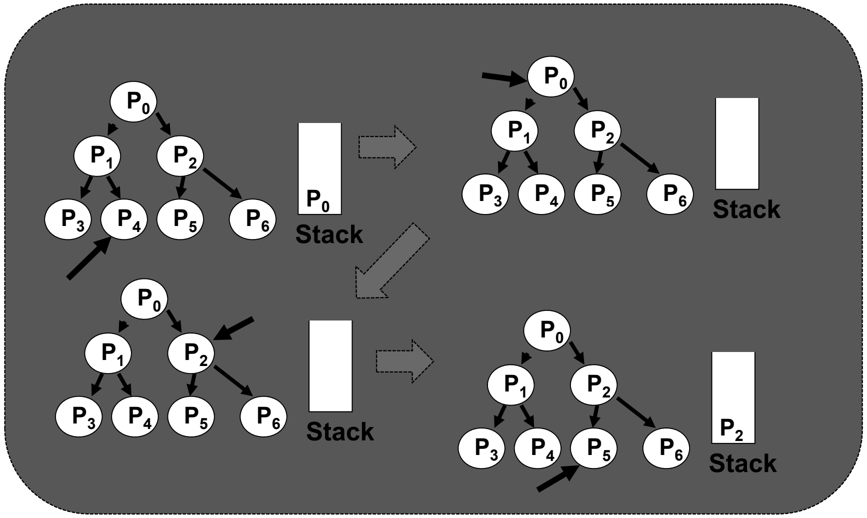
Exhaustive Search

- **Computation can be modeled as state-space search**
  - multiple states and transition from one state to another state
  - one of the states can be a solution state
- **Search space can be modeled as Trees / DAGs**
- **Two major classes of searches: exhaustive search and heuristic search**
- **Exhaustive search potentially examines every state until a solution state is found**
  - Depth-first search traverses left most subtree followed by right, and uses a stack to traverse to right subtree
  - Breadth-first searches level by level left to right, and uses queue
- **Heuristic search uses mathematical equation to prune the search space for a focused efficient search**

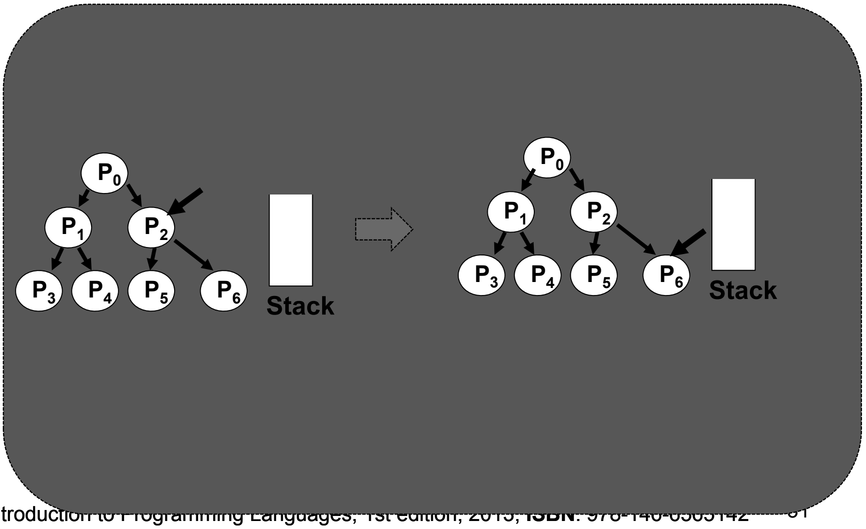
Depth-first Search I



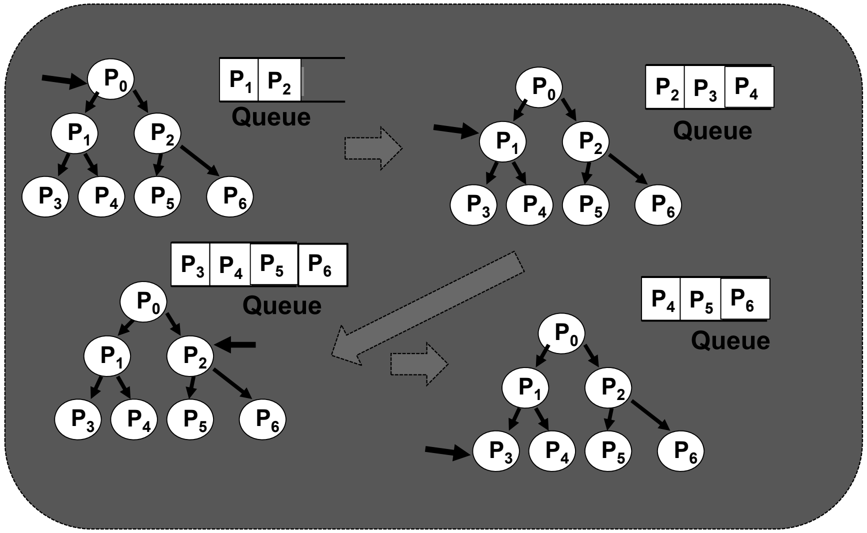
Depth-first Search II



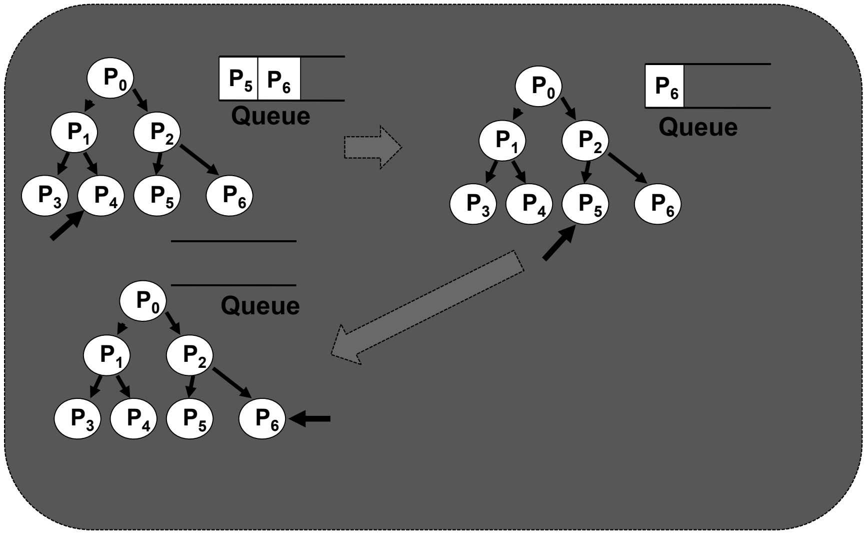
Depth-first Search III



Breadth-first Search I

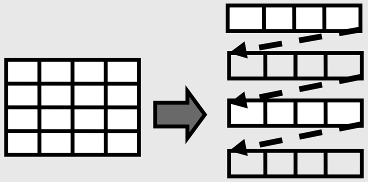


Breadth-first Search II



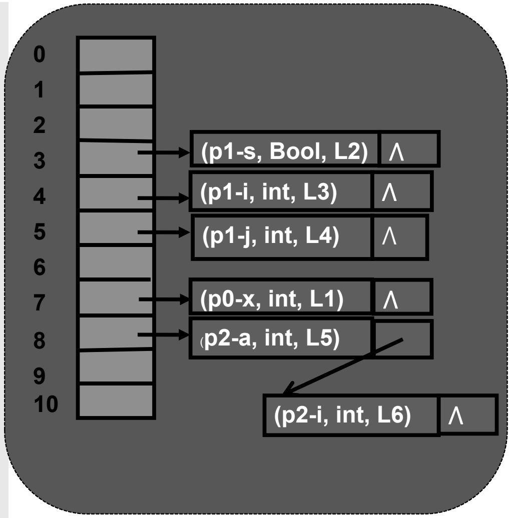
Mapping Data structures

- **Memory is linear**
- **Complex data structures such as structs and multidimensional arrays are mapped to one dimensional array**
- **Two ways to access data-fields**
  - Computational method to compute the offset
  - Reference or pointer based method to access fields
- **Mapping N-dimensional array**
  - Recursively take slices of N-1 dimension, and place them one after another
  - Two dimensional matrix is chain of single dimensional rows/ columns



Hash Tables and Functions

- **Efficient mapping and retrieval of data**
- **Uses hash function on data-content to find the index of stored data**
  - Hash function is simple
  - Uses prime size table to avoid collision
- **Collision is handled using linked lists**
- **Near constant time complexity**



Abstract Computation - I

- **Program is a sequence of meaningful instructions (statements)**
- **Each statement is terminated by a delimiter or linefeed**
- **A program can have**
  - Literals, l-values, r-values, identifiers, labels, definitions, declarations, assignment statement, commands, expressions, procedures and functions, strings, procedure invocations, parameters, and sequencers
  - **Literal –** an elementary expression that can not be further split. Examples: number, character, atom
  - **r-value** – the evaluated value of an expression. Occurs on the right hand side of an assignment. Actual value of a variable
  - **l-value** – location value of a variable. Occurs on the left hand side of an assignment
  - **Identifier** – a symbolic name associated with an entity such as constant, procedure, variable etc.
  - **Definition** – A symbol associated with a value. During compilation symbol is substituted by the corresponding value

Abstract Computation II

- **Variable**
  - identifier → l-value → r-value
  - Can be associated with a concrete value or type
  - When associated with a type is called type variable
  - Can be destructively updated or could be assign-once
- Assignment statement
  - Right hand side expression is evaluated and written into the memory location associated with the variable name.
- **Command is a statement with embedded assignment statement**
- **Expression evaluation does not write into memory location**
- **String is a sequence of characters**
- **Operators could be**
  - Dyadic – having two operands such as addition, subtraction, multiplication, division, logical or, logical and
  - Monadic – having one operand such as not, - *<operand>*

Mutable vs. Assign-once Variables

- **Mutable vs. Assign-once variables**
  - Mutated variables can be destructively updated multiple times
  - Assign-once variables are assigned the value only once

Mutable	Assign-once
1. Reusable 2. Looses past information 3. Undesired program-behaviors due to side-effects	1. Memory explosion 2. Use of past values to find alternate solutions 3. Does not support iteration 4. Less side-effects

---

Binding and Scope Rules

- Binding
  - An entity is associated with corresponding attributes
- **Example**
  - Variable-name bound to a memory location
  - Memory location bound to an r-value
  - Identifier bound to a procedure-block
- **Scope Rule**
  - Defines the visibility of declarations within a part of the programs
  - Can be static or dynamic
  - Static binding means visibility does not change with program execution
  - Dynamic binding means visibility changes with procedure invocation, and unbound variables pick up the value from the declarations in the reverse order of the invoked procedures.

Examples of Scope Rules

Static scope rule

```
main ( )
{   integer  x, y, z;
    x = 4; y = 10; z = 12;
    {integer  temp,  z;
      temp = x; x = y; y = temp; z = 5;}
    print( x, y, z);
}
```

- outer block: x, y, z
- inner-block: temp, z-inner, x, y
- Z-outer is shadowed in the inner block

Dynamic scope rule

```
integer sum(integer  x);
return (x + y);
main ( )
{  {integer  y, z; y = 4; z = 5; sum(y);}
   {integer  w, y, z;
       w = 4, y = 5; z = 6;  sum(z);}
}
```

- first call to sum(y) returns 8; In sum, x gets bound to value of y = 4, and y gets bound to 4.
- Second call sum(z) returns 11. In sum, x gets bound to z = 6, and y gets bound to 5

Types of Variables

- **Variables are classified by visibility rules and lifetime**
  - Global – visible everywhere, lifetime throughout the program
  - Nonlocal – visible in the nested procedures, lifetime is the procedure in which it is declared
  - Local – visible within the procedure they are declared
- **Variables can be static or dynamic**
  - Static variables are allocate memory location at compile time
  - Dynamic variables are allocated memory locations during runtime
- **Variables in object-oriented languages**
  - Class variables – variables declared in class, accesses by all instance of the class
  - Instance variable – only accessible in a specific object

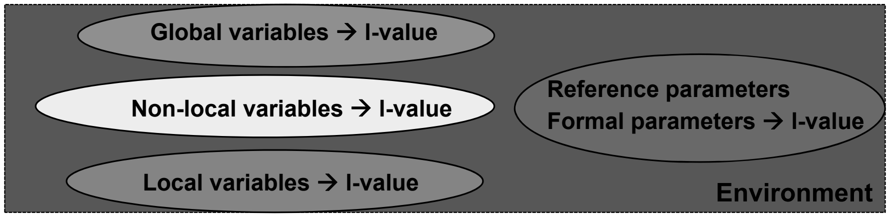
Environment and Store

Environment

- **Environment is set of mapping between identifier and memory locations**
- **Environment changes with a new declaration**
  - Creating new identifier → memory location mapping
  - Shadowing non local variable

Store

- **Store is mapping of memory location to r-value**
- **Store changes with a new assignment statement or initialization or parameter passing**



Global variables → l-value

Non-local variables → l-value

Reference parameters
Formal parameters → l-value

Local variables → l-value

Environment

Functions and Procedures

Function

- **Function is a collection of expressions**
- **Function does not alter the store as it has no destructive updates**
- **Functions has four components**
  - Name, body, parameters and bounded variables

Procedure

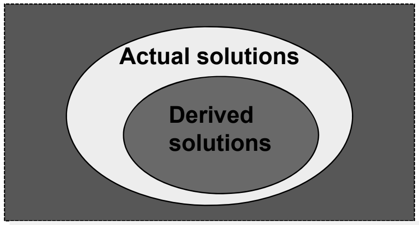
- **Procedure contains atleast one command**
- **Procedure alters the store due to assignment statements**

Abstracting the Program Execution

- **Program execution is modeled as a state transition**
  - Each statement transforms program to a new state.
  - A state is a triple of the form (σ^E, σ^S, σ^D) where σ^E is the environment, σ^S is the store, and σ^D is a stack of pairs of the form (σ^E, σ^S) of the suspended calling procedures in LIFO order.
  - Computational state changes when environment changes, when store changes, when a procedure is called, and when control returns from a procedure
- **Program execution modeled as Boolean state transition**
  - Each state is a Boolean conditions connected through logical operators: logical-and, logical-or and negation
  - Boolean expression changes each time an assignment operation is executed
  - Example: X = 5 + 3 makes X == 8 as true

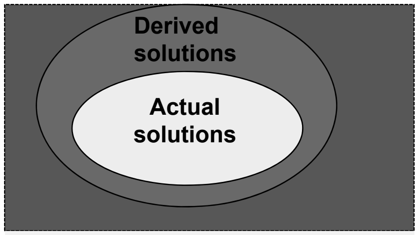
Program Correctness & Completeness

Program Correctness



- Program is correct if the solutions derived by the program is a subset of the actual number of solutions to a problem

Program Completeness



- Program is complete if the actual solutions to a problem is a subset of the number of solutions derived by the program

Program is complete and correct when actual solutions are the same as the derived solutions

---

Principle of Locality

- **A program executes within a small subset of the environment. This subset is called locality**
- **Locality change**
  - Control flow processes different parts of a large data structure
  - Control moves to different parts of a large program
  - Current subprogram calls another subprogram
- **Principle of locality is important because**
  - Only a small part of the environment and store resides into main memory
  - As the locality changes there is an overhead of bringing the needed environment and store into the memory

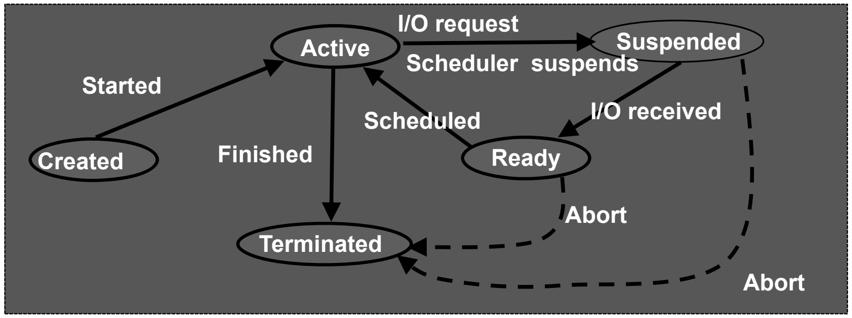
---

Operating System Concepts

- **Middleware – layer between the low level instructions and application**
  - Memory management, system utilities for user friendly interface, I/O management, file management, security and networking
  - Provides access of to multiple processes and users concurrently
- **Operating system concepts related to programming are**
  - Principle of locality, virtual memory, page faults, buffers, processes and threads
- **Process**
  - Active part of a program that is in the machine for execution
  - Is allocated memory blocks to store instructions and data
  - Has a stack, memory area to communicate, and status flags
  - States: executing, ready to execute, sleep, suspended, terminated, waiting for I/O

Thread

- **Thread: a light weight process that shares its memory area with parent process**
  - More efficient than a process due to reduced overhead
- **A process can spawn multiple threads to do subtasks concurrently**



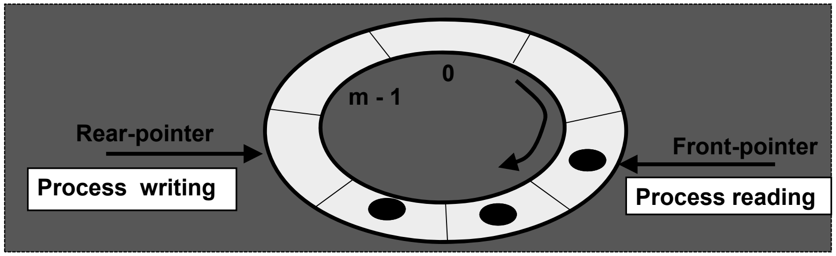
Virtual Memory and Page-faults

- **A programmer works into logical memory space**
  - In a logical memory space, large data structures are contiguous
- **Computer runs into physical memory space that is non-contiguous.**
- Virtual memory maps logical space of the programmer to non-contiguous physical space of the memory
  - Virtual blocks could be fixed size called pages or variable size called segments
  - Page based virtual memory has internal fragmentation but no external fragmentation. If an addressed page is in secondary memory then it is called page fault. Pages are brought from secondary memory to main memory upon page fault, and has significant overhead, and reduces actual CPU utilization.
  - Segmentation has no internal fragmentation. However, bringing a segment may have additional overhead

---

Buffer

- **Definition**
  - Memory space between two asynchronous process so that they can write and read from the memory at their own pace.
- **Implementation: using circular queue**
  - Use of circular buffer allows memory reuse
  - Circular buffer is empty when both front and rear point to the same memory location
  - Circular buffer is full when (rear + 1) mod size = front



Operations on Buffer

- **is_empty_buffer(buffer)**

```
if (fp == rp) return(true)
else return(false);
```

- **is_full_buffer(buffer)**

```
if (((rp + 1) mod m) == fp   return(true)
else return(false);
```

- **insert(buffer, element)**

```
if not (is_full_buffer(buffer)) {
     buffer(rp) = element;
      rp = (rp + 1) modulo m;}
else raise_exception(buffer_full, element)
```

- **remove(buffer)**

```
if not (is_empty_buffer(buffer))
       element = buffer(front-pointer);
        fp= (fp + 1) mod m;
        return(element);}
else raise_exception(buffer_empty)
```

Summary I

- **Four major foundations of programming are**:
  - Data structures, discrete structures, operating system concepts and abstract computational concepts
- **Data structure concepts for programming are**:
  - Stacks, queues, linked lists, trees and exhaustive search
- **Discrete structure concepts for programming are:**
  - Sets, bags, functions, relations, mapping and recursion and FSA
  - Set operations are Cartesian products, union, intersection, disjoint union, power set
  - Ordered bags are important
  - Boolean logic is important for conditions in selection and iteration
- **Operating system concepts for programming are**:
  - Principle of locality, circular buffer, virtual memory

Summary II

- Abstract computation concepts are used to model the program execution abstractly
  - There are many abstract entities such as literal, l-value, r-value, variables, definitions, assignment, expression, command
  - Variable is identifier → l-value → r-value
  - Environment is a set of mapping of identifier → memory locations
  - Store is a set f mapping of memory locations → values
  - An assignment statement destructively updates a memory location
  - A command contains at least one assignment statement
  - An expression does not have an assignment statement
- Scope could be static or dynamic
  - Static scope rule is based upon program structure
  - Dynamic scope rule is based upon the LIFO pattern of the calling procedures