[CSED233-01] Data Structure

Intermediate Summary

Jaesik Park

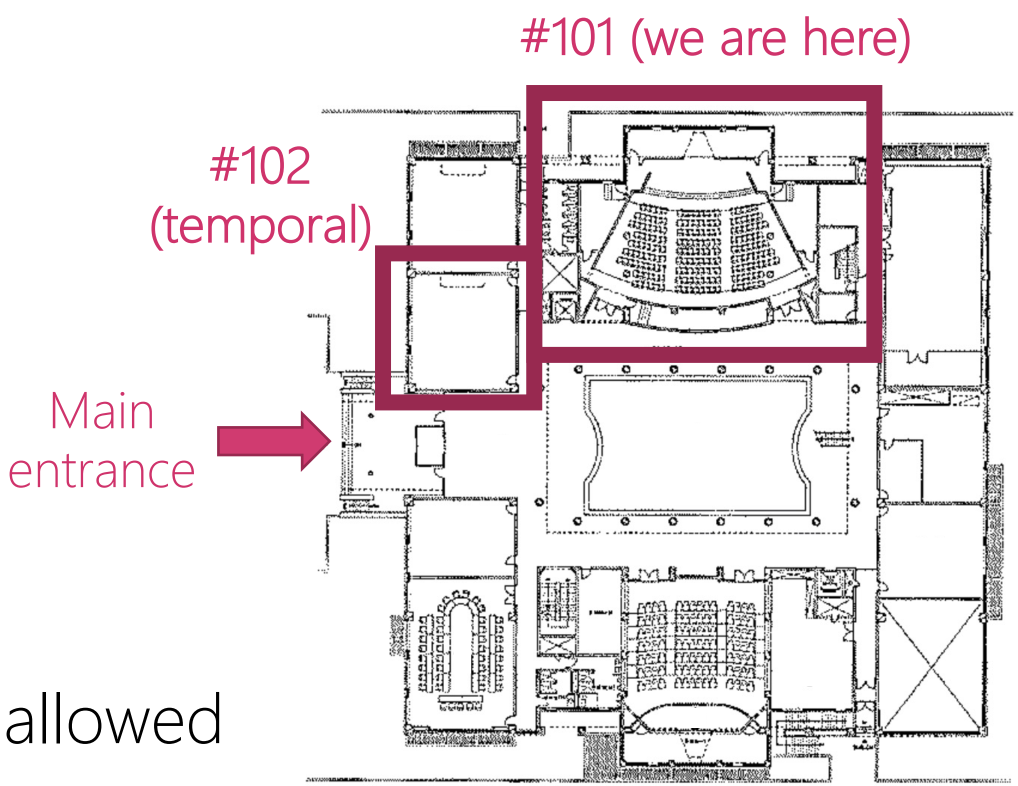
POSTECH

Office Hour

- Not actively used thesedays
  - 6,4,2,1,0,0,0,0,0,…
- Tuesday & Thursday 1~2PM regularly
  → By appointment, available at Tuesday & Thursday 1~2PM, via online

Midterm Exam

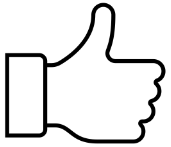
- When: April 11, 8~10AM (Not 9~11AM)
- Where: Elec Bldg (LG동)
  1. #101 Auditorium (Here)
  2. #102 Classroom
  3. The classroom and your seat will be RANDOMLY assigned
- Instructions
  - Bring your pencils and erasers
  - Closed book exam - no electric devices allowed
  - Bring your student ID card
  - Turn off your smart phone



Elec Bldg (LG동)

Topics we have covered

- C++
- Algorithm
- List, stack, queue
- Tree
- Priority queue, heap
- Sorting

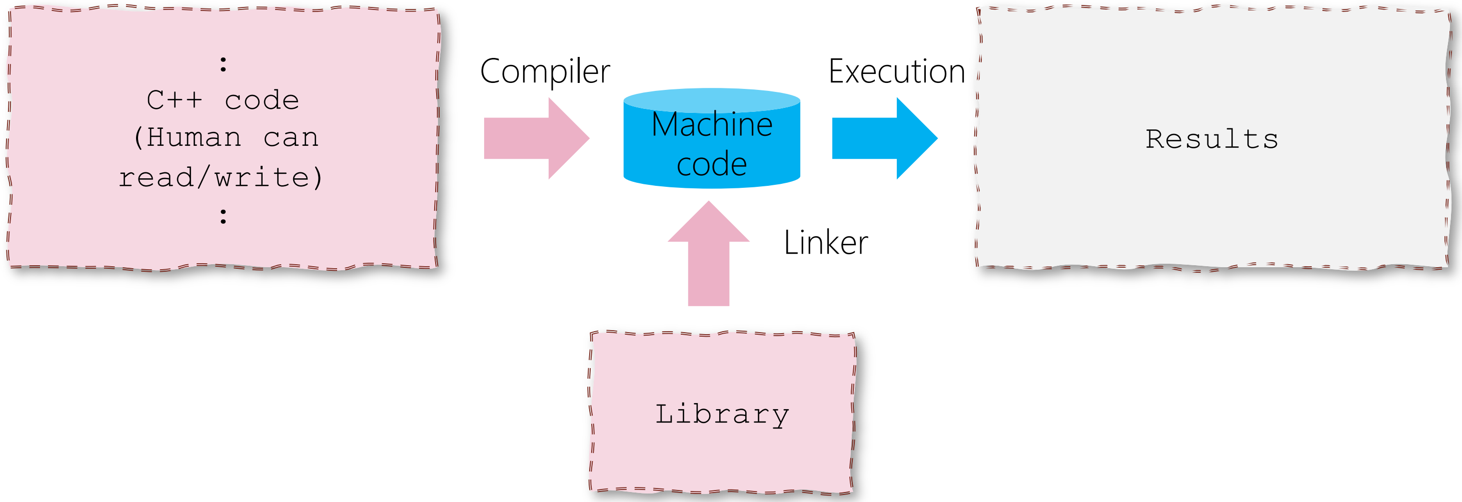
- You have taken 13 lectures and finished two PAs 

C++ Review

- Remind yourself about C++
- Play with C++ and get ready to do assignments

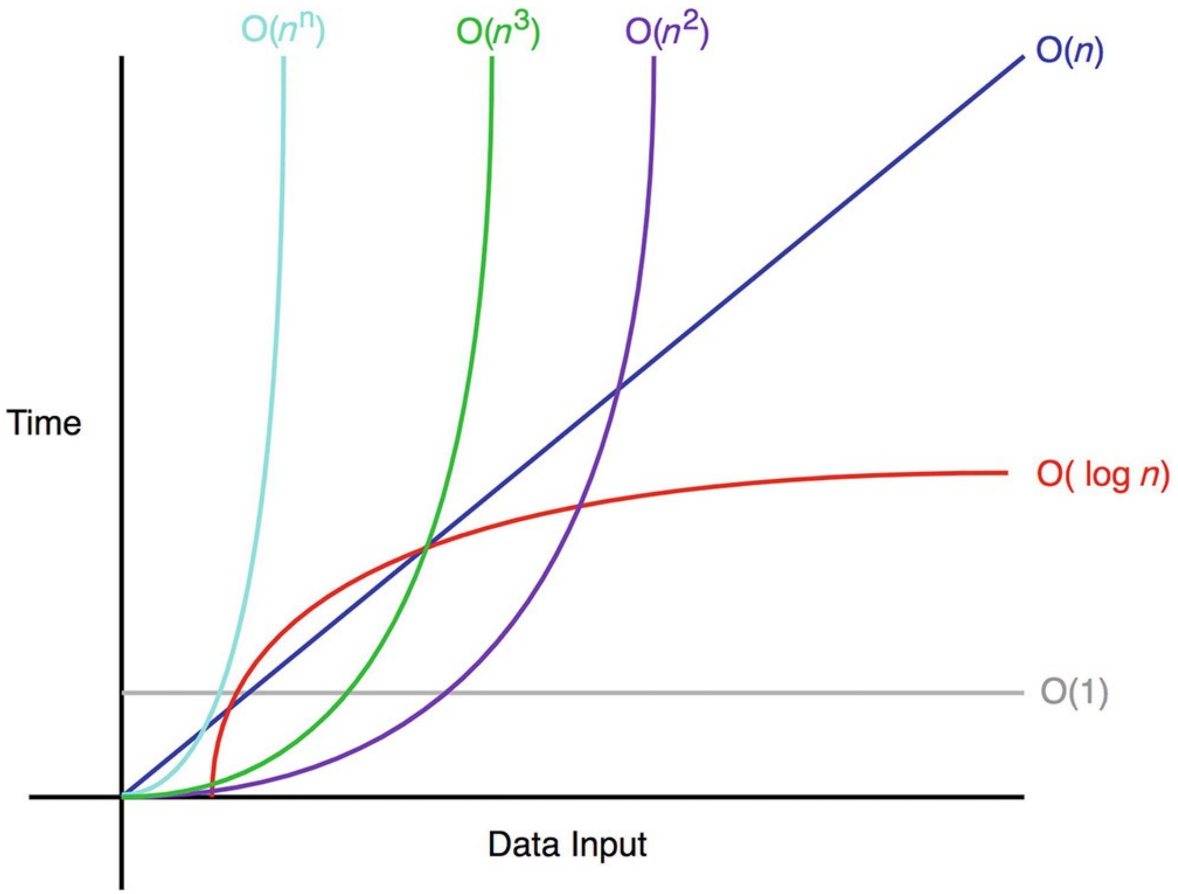
How C++ Works

1. **Create** a C++ source file and **save** it
2. Compile,
   - Using a *compiler*
     - creates a machine-code interpretation of your program
   - Using a *linker*
     - includes any required library functions needed
3. **Execute** program and see results



Algorithm Analysis

- Which algorithm is better regardless of the computer spec. you are using?



Asymptotic Analysis

- Asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior.
  - If $f(n) = n^2 + 3n$, then as n becomes very large, the term 3n becomes insignificant compared to n^2.
  - The function f(n) is said to be "asymptotically equivalent to n^2, as $n \to \infty$".

- Let T(n) the running-time function that maps an input size **N** to a running time **R**

- To capture the growth rate behavior of *T(n)* in the long run
  - Worst case ➔ upper bound: Big-Oh
  - Average case ➔ Equal: Big-Theta
  - Best case ➔ Lower bound: Big-Omega

Asymptotic Algorithm Analysis: Example

program segment

```
for i:=1 to n do
    for j:=1 to n do begin
        C[i,j]:=0;
        for k:=1 to n do
            C[i,j]:=C[i,j]+A[i,k]*B[k,j]
    end
```

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}(c_1 + \sum_{k=1}^{n} c_2) = \sum_{i=1}^{n}\sum_{j=1}^{n}(c_1 + c_2 \cdot n)$$

$$= \sum_{i=1}^{n}(c_1 \cdot n + c_2 \cdot n^2) = c_1 \cdot n^2 + c_2 \cdot n^3$$

$$\Rightarrow T(n) \in O(n^3)$$

Limitations of Analysis

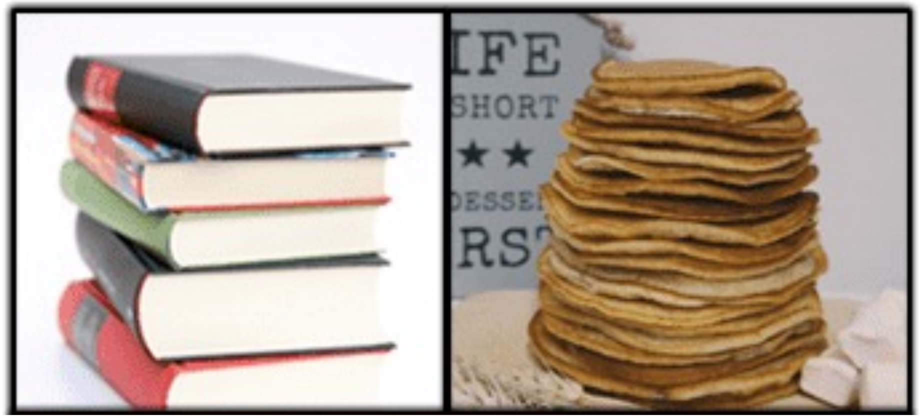
- Not account for *constant factors*, but constant factor may dominate
  - $1000*n$ vs. n^2 (when interested only in $n < 1000$)

- Not account for different *memory access times* at different levels of memory hierarchy
  - Cache Memory << MM << HDD

- Programs that do more computation may take less time than those that do less computation
  - Cost (fetch from *MM*) >> Cost (operation in *CPU*)
  - Memory access could take more than computation

List, Stack, Queue

- List, stack, queue “something” as the names suggest ☺



(Linear) Lists

- List $L = <a_1, a_2, \ldots, a_n>$
  - a finite, *ordered* collection of elements
  - n: length (size) of the list
    - empty list < > : $n = 0$ (no elements)

- Position of a_i is i
  - head (front) ↔ tail (rear)
  - "*current*" position

    <20, 23, | 12, 15> : separated by *fence*

    <20, 23, | 10, 12, 15> after insertion of 10 (at "current" position)

head tail

$<a_1, a_2, \ldots, a_n>$

- Sorted list ↔ Unsorted list
- Don't be confused, ordered and sorted mean different things

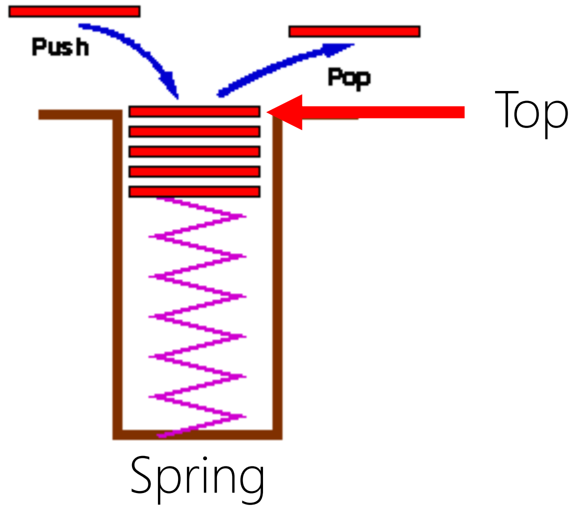
Stacks

- All insertions & deletions take place at one end (called *Top*)
- Special type of list
  - LIFO (Last-In, First-Out)
    - Pushdown list
- You can implement stacks using any type of list implementation (pointer, array, cursor, …)

- $S = <a_1, a_2, \ldots, a_n>$

  $\uparrow$ *Top*

  - *Push(x, S)* $S = <x, a_1, a_2, \ldots, a_n>$
  - *Pop(S)* $S = <a_2, \ldots, a_n>$
  - *Top(S)* a_1
  - *MakeNull(S)* $S = <>$
  - *IsEmpty(S)* true if $S = <>$



Queues

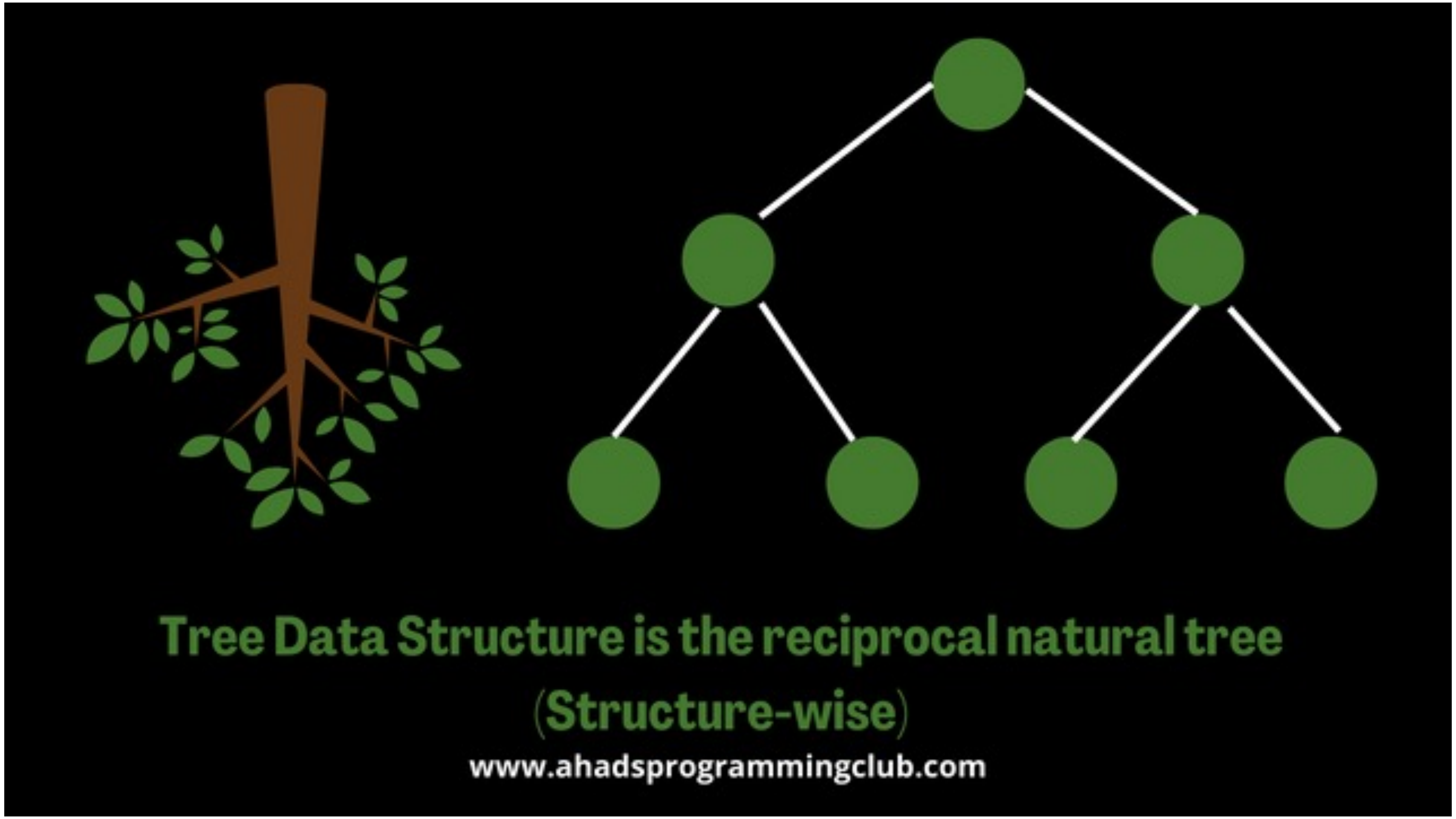
- FIFO (First-In First-Out) list
- Similar to top in the stack, here we have front and rear
- $Q = \langle a_1, a_2, \ldots, a_n \rangle$

$$Q = \langle \underset{\substack{\uparrow \\ \textit{Front}}}{a_1}, a_2, \ldots, \underset{\substack{\uparrow \\ \textit{Rear}}}{a_n} \rangle$$

  - *Enqueue(x, Q)* $\quad Q = \langle a_1, a_2, \ldots, a_n, x \rangle$
  - *Dequeue(Q)* $\quad Q = \langle a_2, a_3, \ldots, a_n \rangle$
  - *Front(Q)* $\quad a_1$
  - *MakeNull(Q)* $\quad Q = \langle\, \rangle$
  - *IsEmpty(Q)* $\quad$ true if $Q = \langle\, \rangle$

Tree

- Tree data structure looks like a (upside-down) tree
  - Family tree, academic tree, …

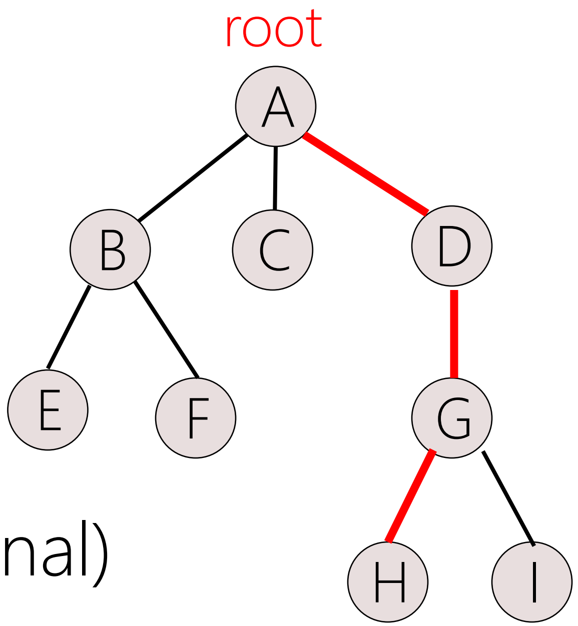


Parenthood Relations

- Parent/child: A is the parent of C, C is the child of A
- Ancestor/descendant: A is the ancestor of G, G is the descendant of A
- Siblings: B,C,D are siblings

- Path $<n_1, n_2, \ldots, n_k>$
  - n_i is the parent of n_{i+1} $(1 \leq i < k)$
  - length = k -1
    - Number of edges connecting the path

- Terminal (leaf, external) ↔ Non-terminal (internal)
  - Whether the node has any child nodes
  - E,F,C,H,I ➔ leaf
  - B,A,D,G ➔ Non-terminal



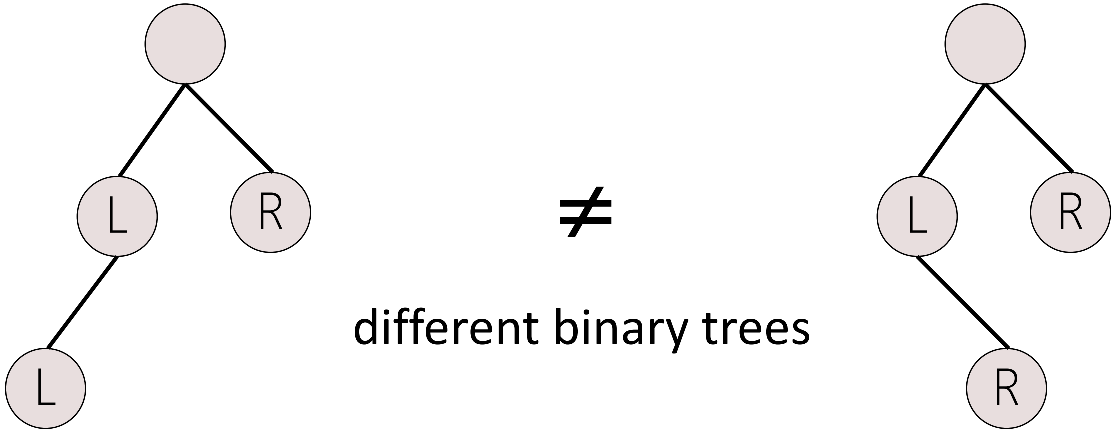
Path from A to H

(length = 3)

Binary Trees

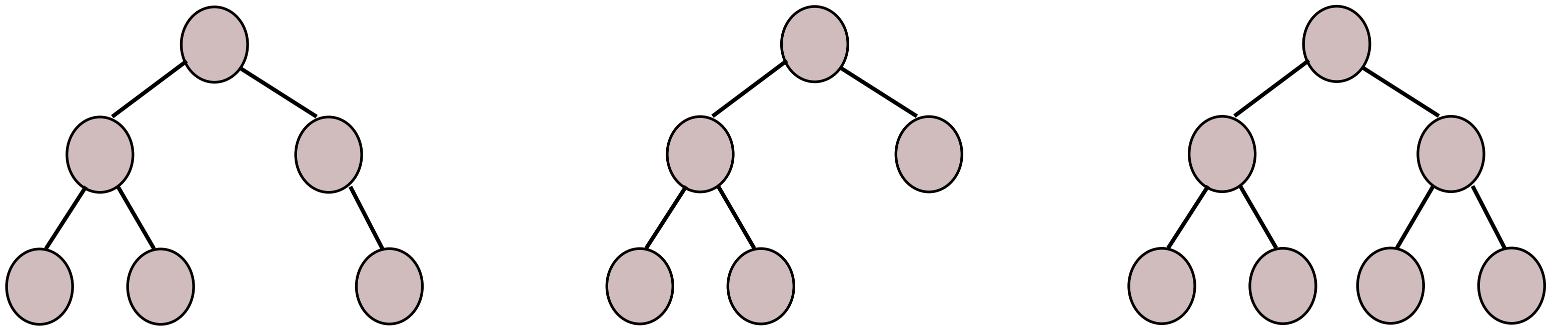
- Binary tree
  - Every node has at most two children
  - Each child is designated as a left child or a right child

- Examples:



Proper, Full Binary Trees

- Proper
  - if each node has either zero or two children
- Full
  - If it has a maximum # of nodes at each level
  - A full binary tree of height h has $(2^{h+1} - 1)$ nodes



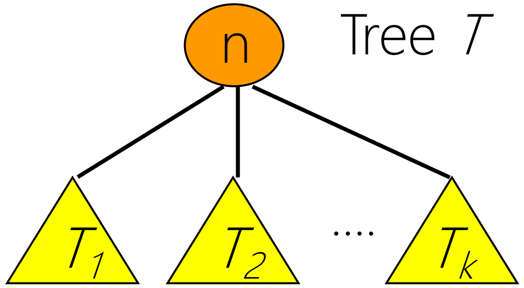
Height = 2

proper (not full) *full*

- Caution: *different definitions* in some texts
  - In our text, proper binary trees are also known as full binary trees

Recap: Tree Traversal

- Types of traversals



- $Preorder(T) = \langle n,\ Preorder(T_1), \ldots, Preorder(T_k) \rangle$
- $Postorder(T) = \langle Postorder(T_1), \ldots, Postorder(T_k),\ n \rangle$
- $Inorder(T) = \langle Inorder(T_1),\ n,\ Inorder(T_2), \ldots, Inorder(T_k) \rangle$
    - No natural definition of *Inorder*

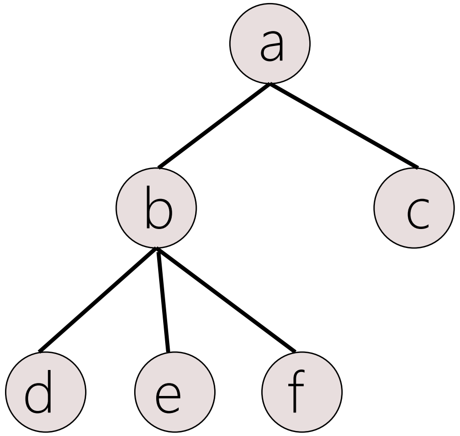
Unique Binary Tree (by Two Traversals)

- We can identify the binary tree uniquely by two traversal sequences like:
  - (*postorder* & *inorder*), (*preorder* & *inorder*), (*level-order* & *inorder*)
    - *inorder*: to find Left & Right child/subtrees
    - *postorder*: to find the Root (the last in *postorder*)
    - *preorder*: to find the Root (the first/ in *preorder*)
    - *level-order*: to find the Root

- However, the other combinations leaves some ambiguity in the tree structure

General Tree Implementations

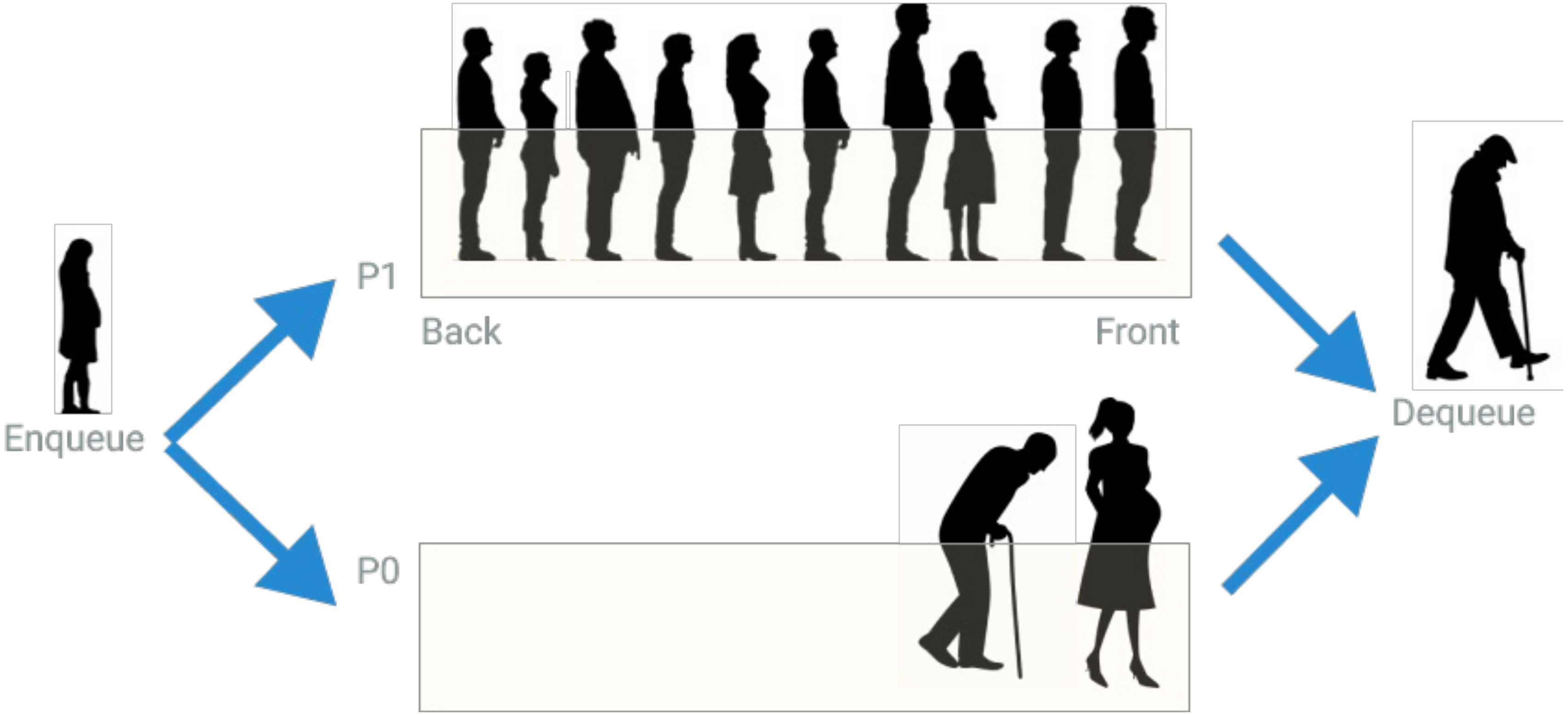
- General k-ary tree is a tree in which each node has no more than k children
- Implementations
  - Simple array implementation
  - *List-of-Children* implementation
  - *Left-Child/Right-Sibling* implementation
  - …



- Again, there would be other implementations as well
- No rules here, they have their own pros/cons
- Let's analyze them now

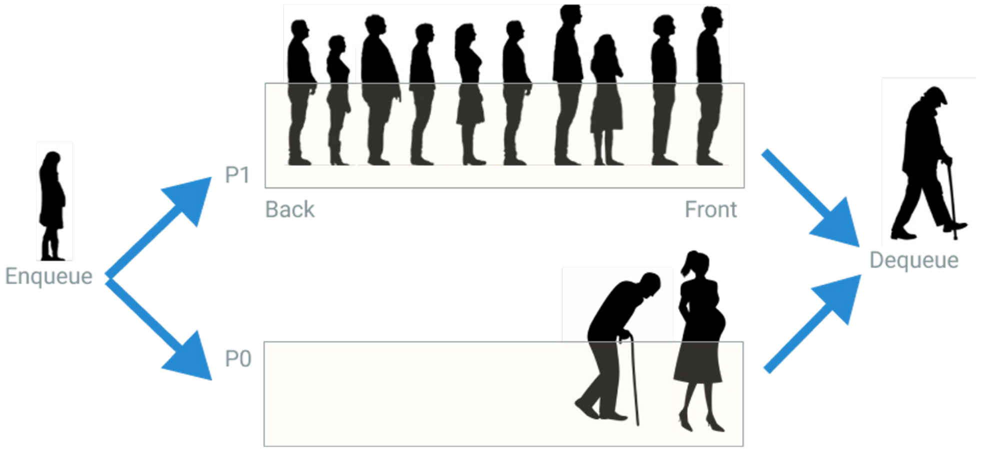
Priority Queue, Heap

- Priority does matter in our life



Priority Queue

- We learned queue
  - FIFO (First-In First-Out) list
  - Similar to top in the stack, here we have front and rear
  - $Q = <a_1, a_2, \ldots, a_n>$
  - Enqueue, Dequeue, …
- How to incorporate priority for queue?
- Priority queue consists of a set of elements (organized by *priority*)
  - Each element *x* has a *priority p(x)* (also called *importance* or *key*)
    - Not necessarily unique
  - Supports the following operations:
    - *Insert(x, H)* – arbitrary element insertion
    - *DeleteMin(H)* = *Min(H)* + *Delete*
      - Delete elements in the order of priority

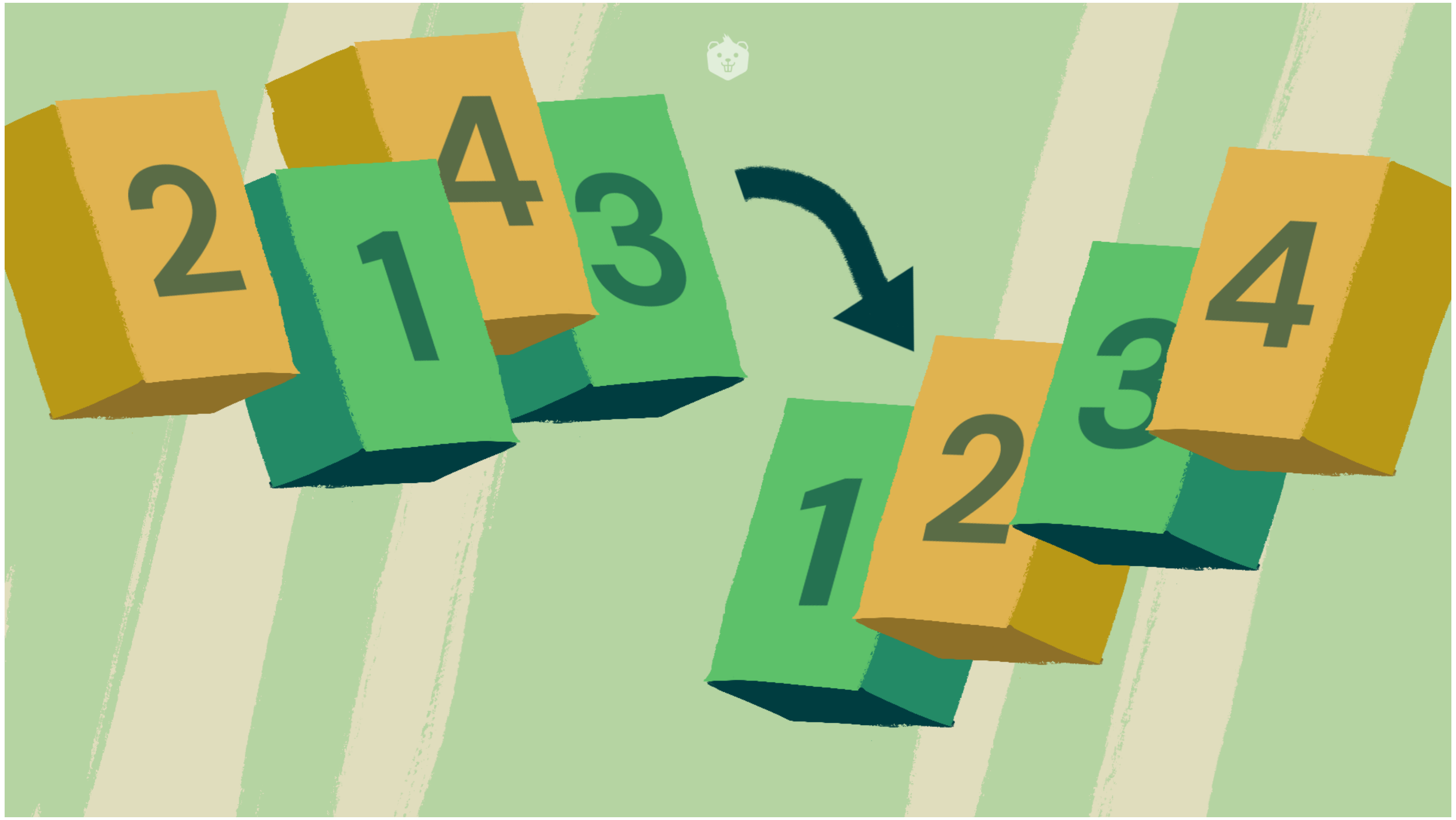


Heap

- Tree-based data structure that satisfies the *heap property*
  - *if B is a child node of A, then* $p(A) \leq p(B)$
  - Implies that an element with the lowest priority is always in the root node (*min-heap*) ↔ *max-heap*
- To efficiently implement a priority queue
  - *Insert* & *DeleteMin*: $O(\log n)$
- There are different types of heaps
  - Binary heap
  - Binominal heap
    - Supports quickly merging two heaps
  - Fibonacci heap, 2-3 heap, etc.
- We will learn binary heap as an example ☺

Sorting

- How to computationally make things ordered?



Types of Sorting: Memory Usage

- Internal (In-memory) sort
  - Appropriate for sorting a collection of elements that fit in main memory
  - Simple, but relatively slow - $O(n^2)$
    - *bubble sort, selection sort, insertion sort*
  - Considerably better - $O(n \log n)$ on average
    - *heap sort, quick sort, merge sort*

- External sort
  - Too large collection of elements to fit in main memory, so the records must reside in external memory
  - Based on *merge sort*

Comparing Quick & Heap Sorts

n	Quick sort		Heap sort		Insertion sort	
	Compare	Exchange	Compare	Exchange	Compare	Exchange
100	712	148	2,842	581	2,596	899
200	1,682	328	9,736	1,366	10,307	3,503
500	5,102	919	53,113	4,042	62,746	21,083

- Heap sort? (heapify!)
- Empirically, quick sort is considerably faster than heap sort
- However, quick sort should never be used in applications which require a guarantee of response time unless it is treated as an $O(n^2)$ algorithm

Remaining topics

- Binary Search Tree
- AVL tree
- B-tree
- Dictionary
- Hashing
- Graph representation
- Graph traversal
- Shortest path finding
- Minimum spanning tree

- Two more PAs and the Final Exam left

Remark

- Thanks for your all the hard work!
- Data structure is essential for many fields in computer science
    - Concept
    - Programming
    - Applications
    - https://ecse.postech.ac.kr/research-activities/research-field/

- Professors
    - https://ecse.postech.ac.kr/member/professor/