Elementary Data Structures

Stacks & Queues Lists, Vectors, Sequences
Amortized Analysis Trees

Abstract Data Types (ADTs)

An abstract data type (ADT) is an abstraction of a data structure

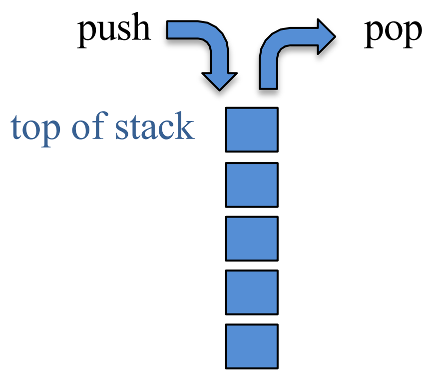
An ADT specifies:

- Data stored
- Operations on the data
- Error conditions associated with operations

Example: ADT modeling a simple stock trading system

- The data stored are buy/sell orders
- The operations supported are
  - order buy(stock, shares, price)
  - order sell(stock, shares, price)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

Stack ADT

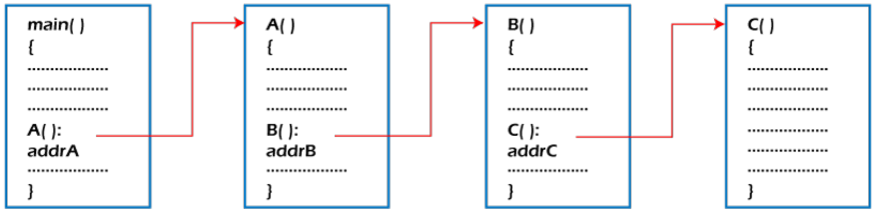


- Container that stores arbitrary objects
- Insertions and deletions follow last-in first-out (LIFO) scheme

- Main operations
  - push(object): insert element
  - object pop(): remove and returns last element
- Auxiliary operations
  - object top(): returns last element without removing it
  - integer size(): returns number of elements stored
  - boolean isEmpty(): returns whether no elements are stored

Applications of Stacks

- Direct
  - Page visited history in a web browser
  - Undo sequence in a text editor
  - Chain of method calls in C++ runtime environment(Recursion)



Function call

- Indirect
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- Add elements from left to right in an array S of capacity N
- A variable t keeps track of the index of the top element
- Size is $t+1$
- Overflow & underflow

Algorithm *push*(*o*):
  if $t = N\text{-}1$ **then**
    throw *FullStackException*   *//overflow*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

$O(1)$

Algorithm *pop*():
  if *isEmpty*() **then**
    throw *EmptyStackException*   *//underflow*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

$O(1)$

Extendable Array-based Stack

- In a push operation, when the array is full, we can replace the array with a larger one instead of throwing an exception
  - Values in old array must be copied over to the new array
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

$N^* = ?$

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

Comparing the Strategies via Amortization

- Amortization: analysis tool to understand running times of algorithms that have steps with widely varying performance

- We compare incremental vs. doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations in the stack.

- We call amortized time of a push operation the average time taken by a push over a series of operations
  - i.e., $T(n) / n$

- Assume we start with an empty stack represented by an empty array

Incremental Strategy

- We replace the array $\boldsymbol{k} = \boldsymbol{n}/\boldsymbol{c}$ times

•Starting with stack of size 0, If we perform **n** pushes in total, and the stack is extended every c pushes.

- Total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of n push operations is proportional to:

$$
\begin{aligned}
& \boldsymbol{n} + \boldsymbol{c} + 2\boldsymbol{c} + 3\boldsymbol{c} + 4\boldsymbol{c} + \ldots + \boldsymbol{kc} \\
= \;& \boldsymbol{n} + \boldsymbol{c}(1 + 2 + 3 + \ldots + \boldsymbol{k}) \\
= \;& \boldsymbol{n} + \boldsymbol{ck}(\boldsymbol{k} + 1)/2
\end{aligned}
$$

- Since $\boldsymbol{c}$ is constant, $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{k}^2)$, which is $\boldsymbol{O}(\boldsymbol{n}^2)$

- The amortized time of a push operation is $\boldsymbol{O}(\boldsymbol{n})$

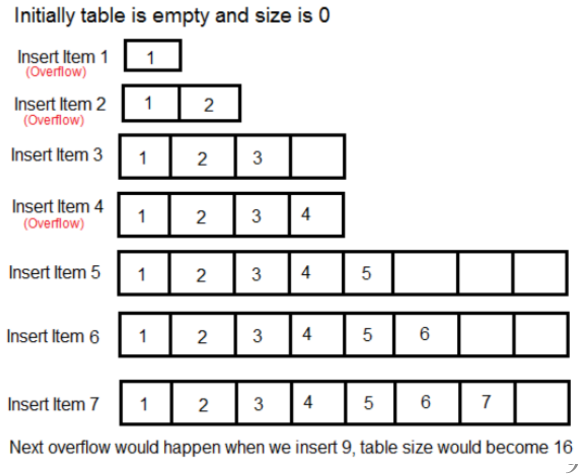
Doubling Strategy

- We replace the array $\boldsymbol{k} = \log_2 \boldsymbol{n}$ times
- Total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of n push operations is proportional to:

$$\boldsymbol{n} + 1 + 2 + 4 + 8 + \ldots + 2^{\boldsymbol{k}}$$
$$= \boldsymbol{n} + 2^{\boldsymbol{k}+1} - 1$$
$$= \boldsymbol{n} + 2^{\log \boldsymbol{n} + 1} - 1$$
$$= \boldsymbol{n} + 2^{\log \boldsymbol{n}}\, 2^1 - 1$$
$$= \boldsymbol{n} + \boldsymbol{2n} - 1$$
$$= 3\boldsymbol{n} - 1$$

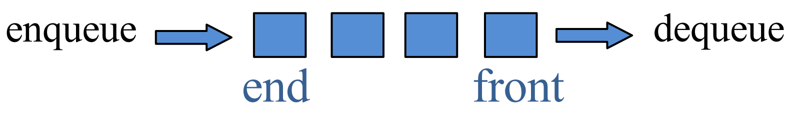
Recall the summation of this geometric series:

$$2^0 + 2^1 + .. + 2^k = 2^{k+1} - 1$$



- $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n})$
- The amortized time of a push operation is $\boldsymbol{O}(\boldsymbol{1})$

Queue ADT



- Container that stores arbitrary objects
- Insertions and deletions follow first-in first-out (FIFO) scheme

- Main operations
  - enqueue(object): insert element at end
  - object dequeue(): remove and returns front element
- Auxiliary operations
  - object front(): returns front element without removing it
  - integer size(): returns number of elements stored
  - boolean isEmpty(): returns whether no elements are stored

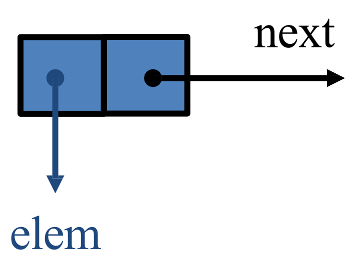
Applications of Queues

- Direct
  - Waiting lines
  - Access to shared resources
  - Multiprogramming

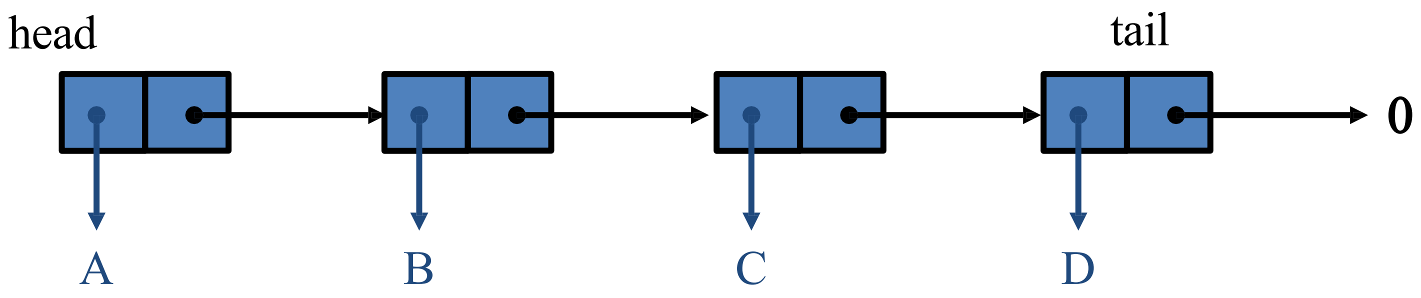
- Indirect
  - Auxiliary data structure for algorithms
  - Component of other data structures

Singly Linked List

- A data structure consisting of a sequence of nodes

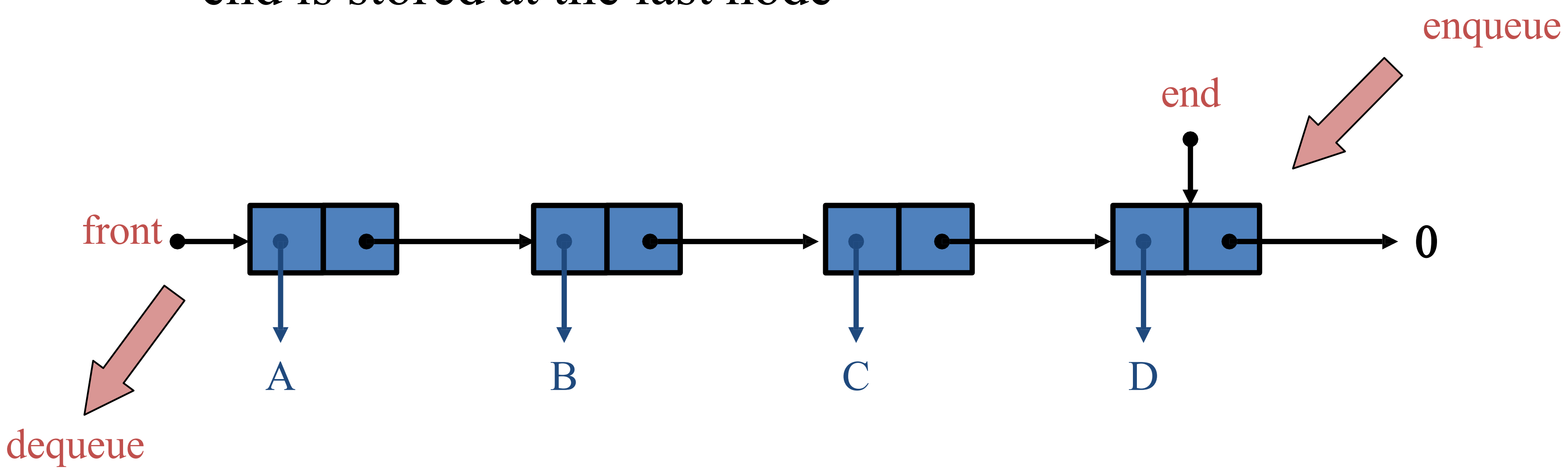


- Each node stores an element and a link to the next node



Queue with a Singly Linked List

- Singly Linked List implementation
  - front is stored at the first node
  - end is stored at the last node



- Space used is $\boldsymbol{O(n)}$ and each operation takes $\boldsymbol{O}(1)$ time

List ADT

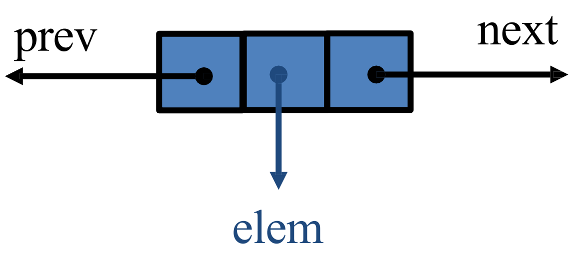
- A collection of objects ordered with respect to their **position** (the node storing that element)
  - each object knows who comes before and after it
- Allows for insert/remove in the "middle"

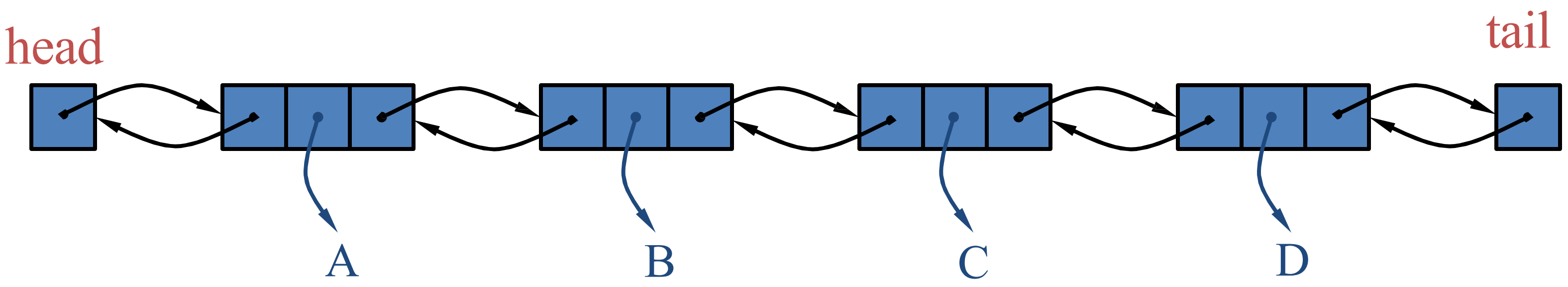
- Query operations
  - isFirst(p), isLast(p)
- Accessor operations
  - first(), last()
  - before(p), after(p)

- Update operations
  - replaceElement(p, e)
  - swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e)
  - insertFirst(e), insertLast(e)
  - remove(p)

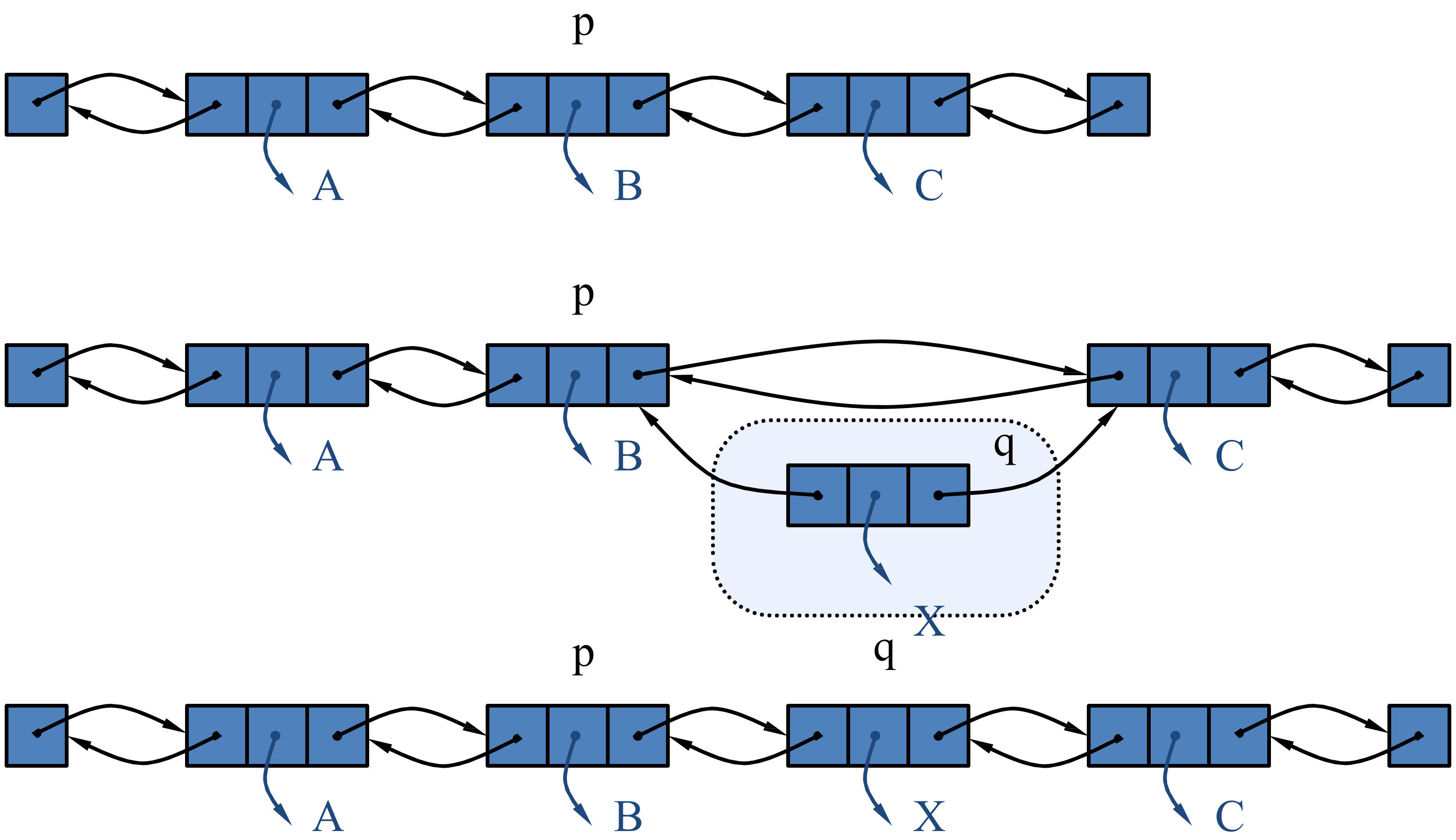
Doubly Linked List

- Provides a natural implementation of List ADT
- Nodes implement position and store
  - element
  - link to previous and next node
- Special head and tail nodes



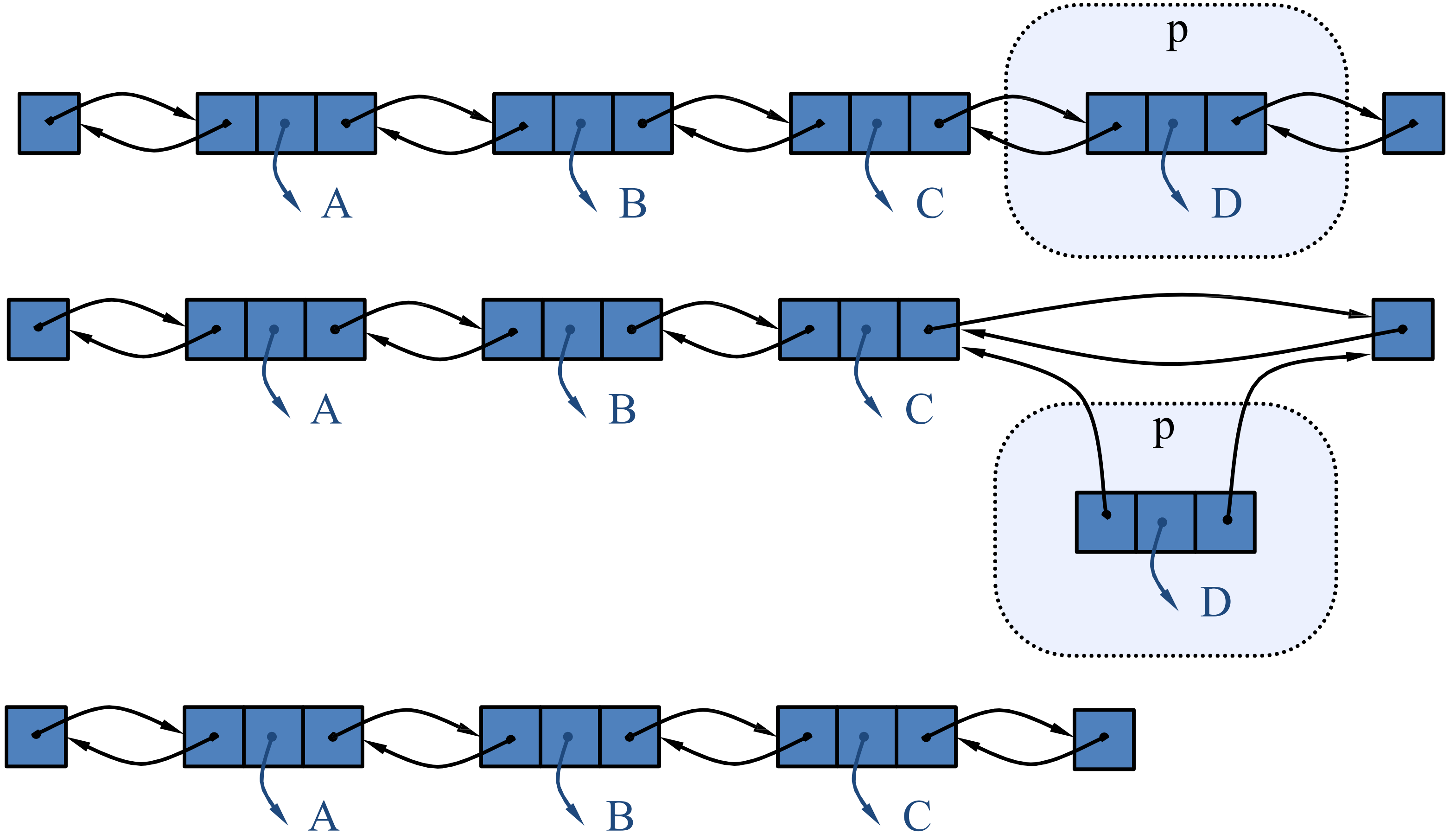


Insertion: insertAfter(p, X)



Deletion: remove(*p*)

- We visualize remove(p), where p = last()



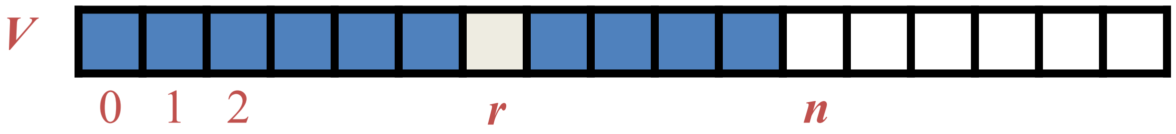
Vector ADT

- A linear sequence that supports access to its elements by their **rank** (number of elements preceding it)

- Main operations:
  - size()
  - isEmpty()
  - elemAtRank(r)
  - replaceAtRank(r, e)
  - insertAtRank(r, e)
  - removeAtRank(r)

Array-based Vector

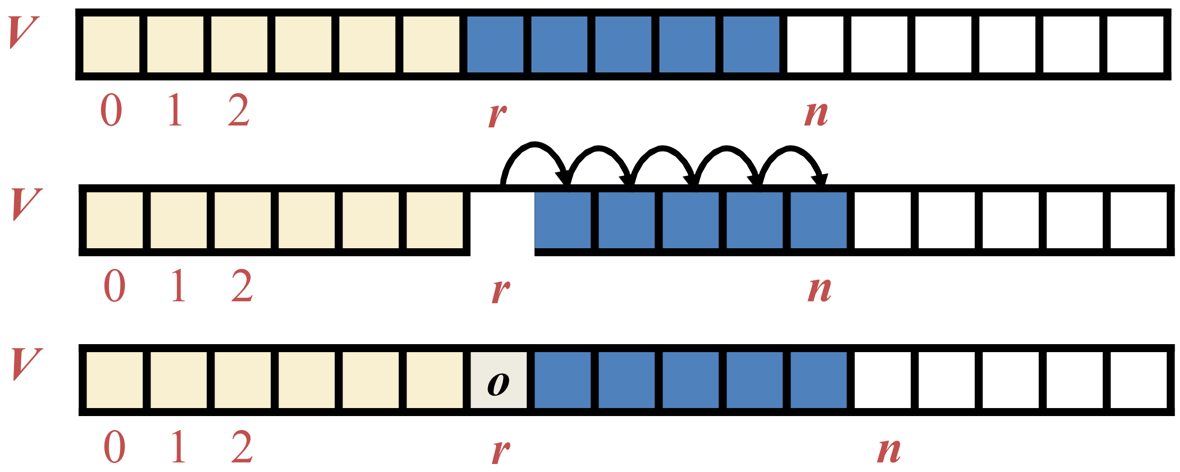
- Use an array *V* of size *N*
- A variable ***n*** keeps track of the size of the vector (number of elements stored)
- ***elemAtRank***(***r***) is implemented in ***O***(1) time by returning ***V***[***r***]



Insertion: insertAtRank(r, o)

- Need to make room for the new element by shifting forward the $n - r$ elements $V[r]$, …, $V[n - 1]$

- In the worst case ($r = 0$), this takes $O(n)$ time

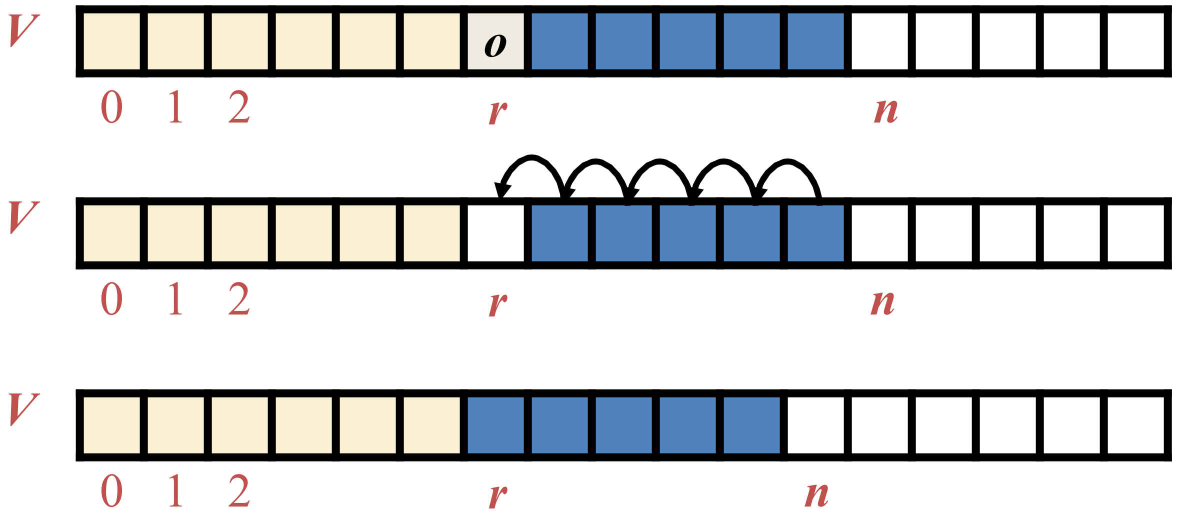


- We could use an extendable array when more space is required

Deletion: removeAtRank(r)

- Need to fill the hole left by the removed element by shifting backward the $\boldsymbol{n} - \boldsymbol{r} - 1$ elements $\boldsymbol{V}[\boldsymbol{r} + 1], \ldots, \boldsymbol{V}[\boldsymbol{n} - 1]$

- In the worst case ($\boldsymbol{r} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time



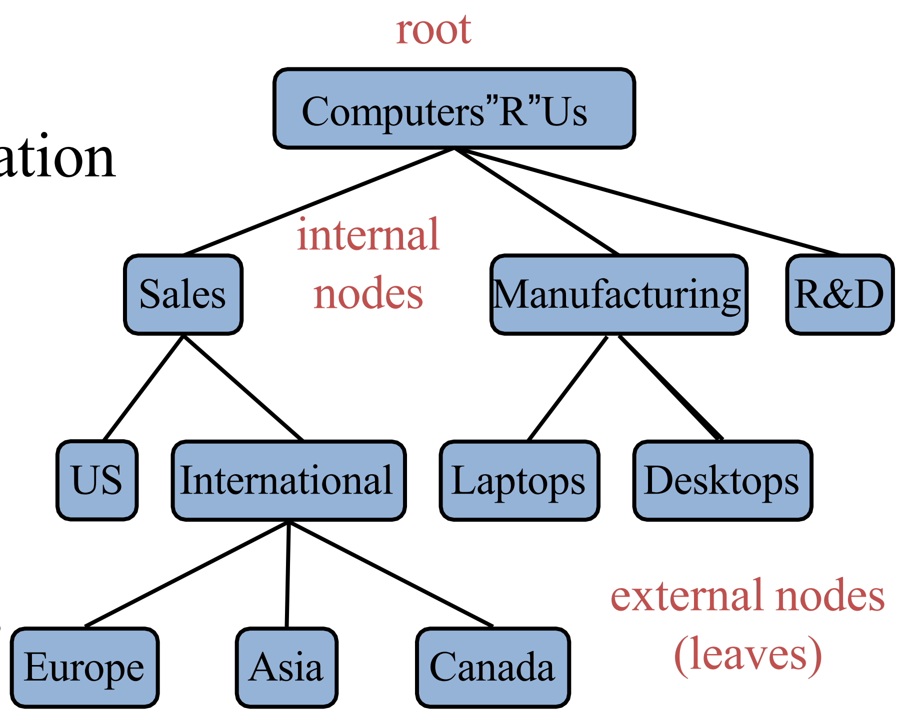
Sequence

- A generalized ADT that includes all methods from vector and list ADTs
- Provides access to its elements from both rank and position
- Can be implemented with an array or doubly linked list

Operation	**Array**	**List**
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$\boldsymbol{O(1)}$	$O(n)$
first, last, before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$\boldsymbol{O(1)}$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$\boldsymbol{O(1)}$
remove (at given position)	$O(n)$	$\boldsymbol{O(1)}$

Tree

- Stores elements hierarchically
- Each node has a parent-child relation
  - Direct applications:
    - Organizational charts
    - File systems
    - Programming environments

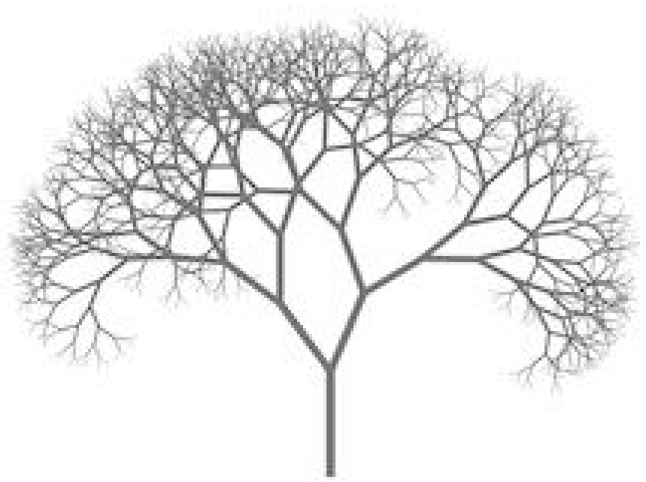


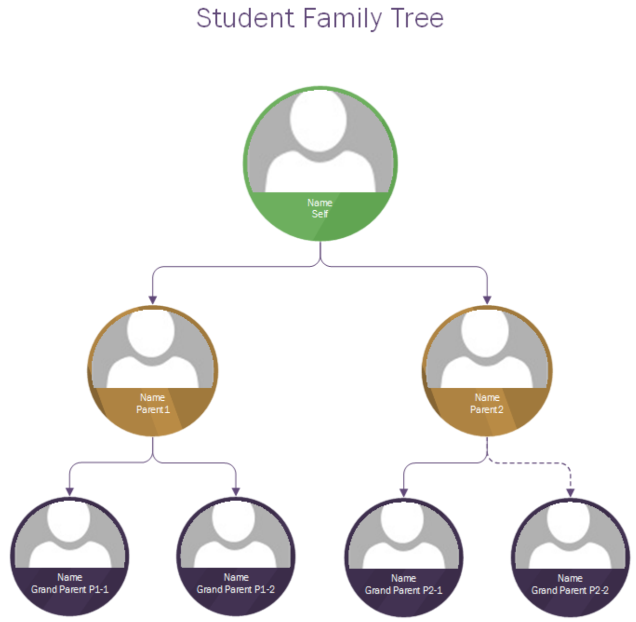
Binary Trees

In a binary tree, each node has two subtrees

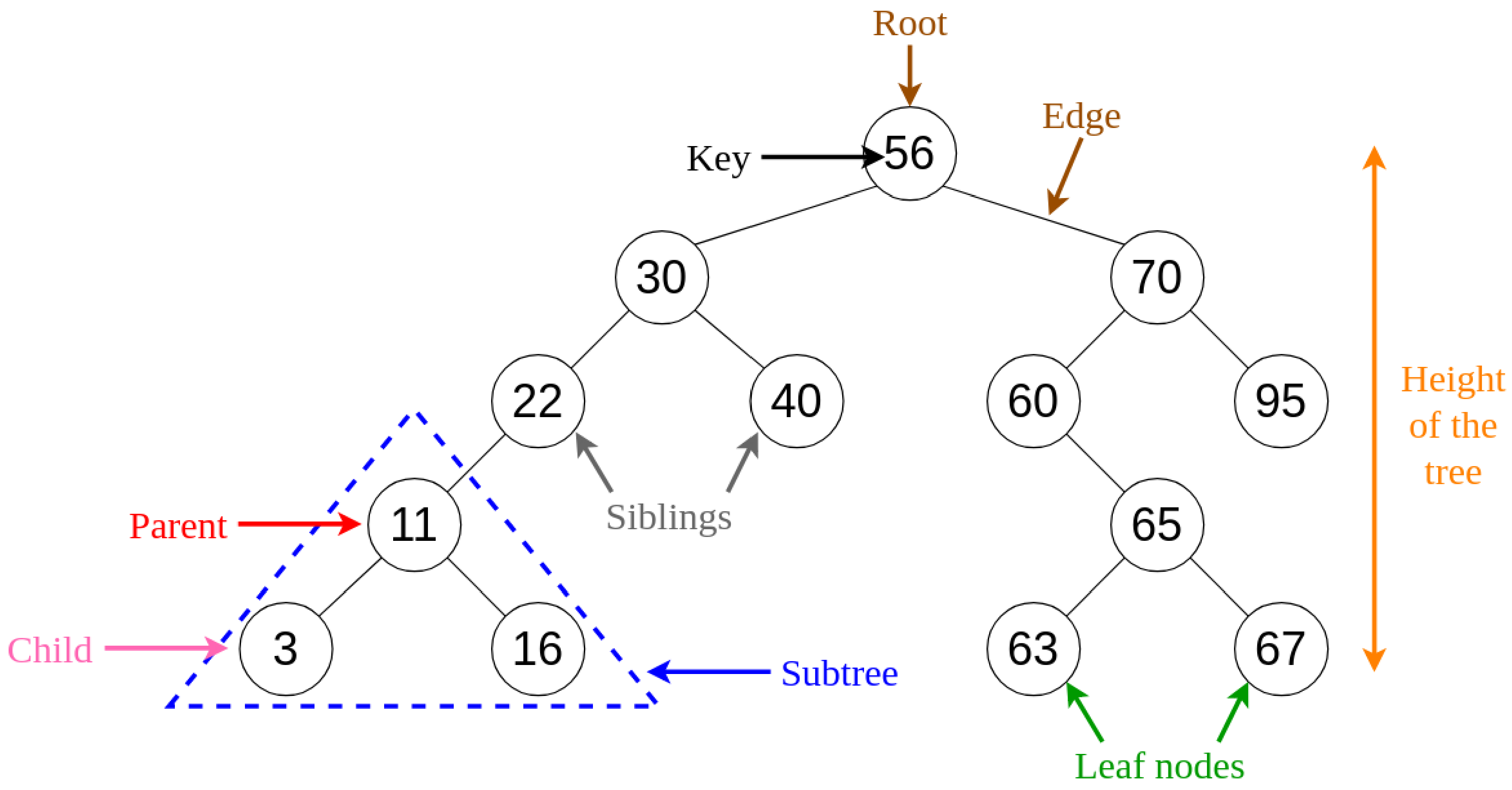
A set of nodes T is a binary tree if either of the following is true

- T is empty
- Its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary trees
- (T_L = left subtree; T_R = right subtree)



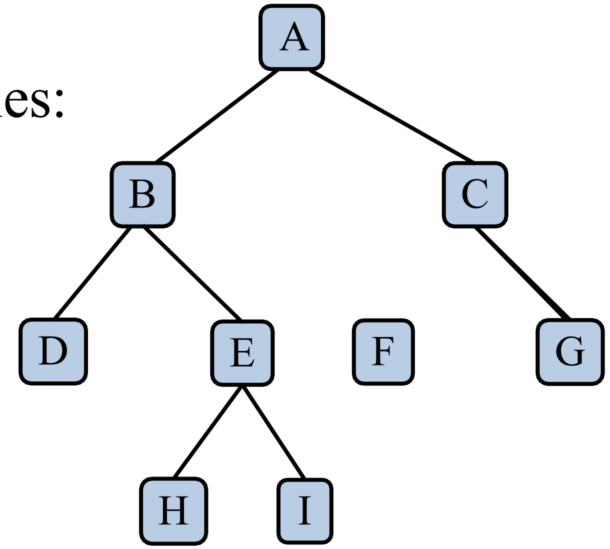


Tree Terminologies



source

(Full) Binary Trees

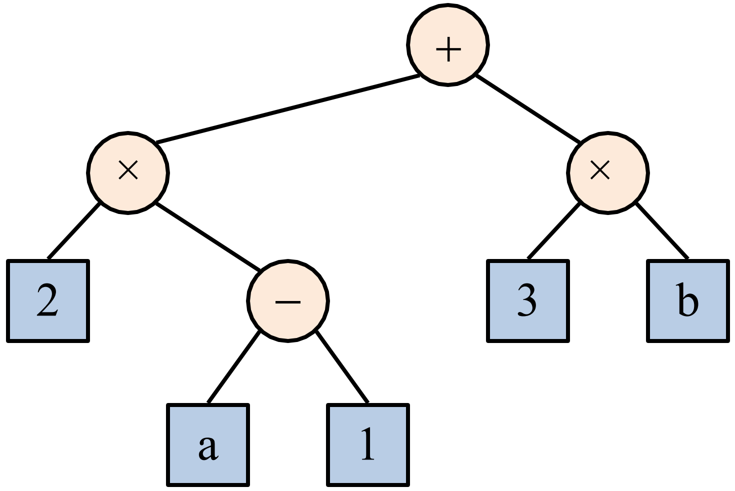


- A binary tree is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair (left child, right child)
- Recursive definition: a binary tree is
  - A single node is a binary tree
  - Two binary trees connected by a root is a binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching

Arithmetic Expression Tree

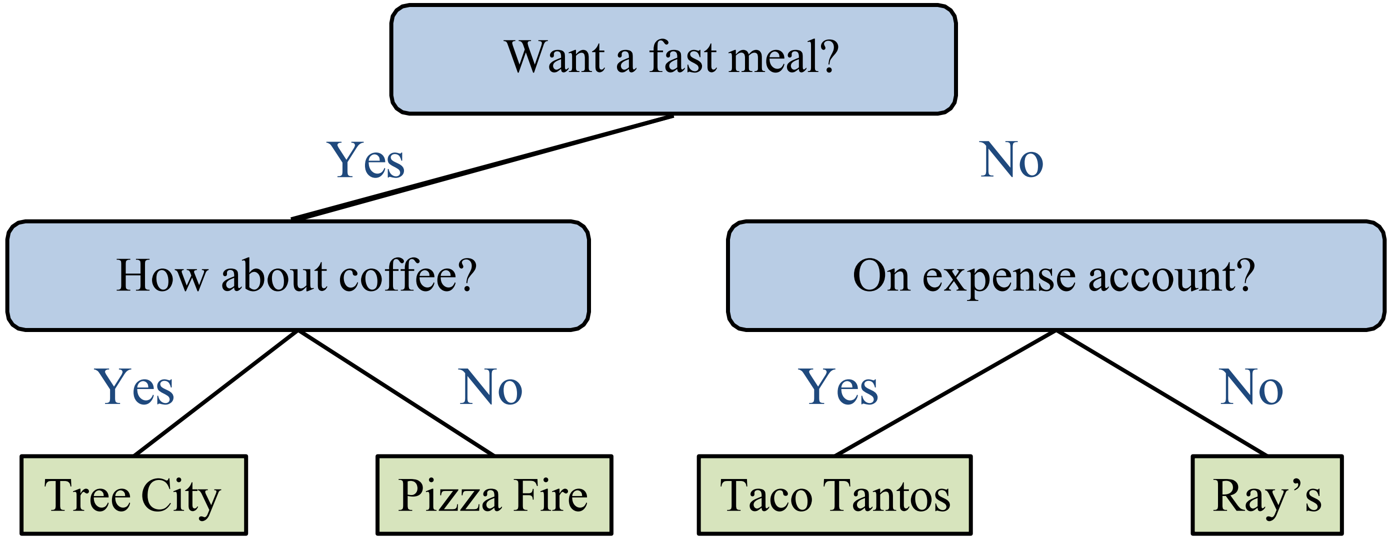
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands

- Ex: arithmetic expression tree for expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Ex: dining decision

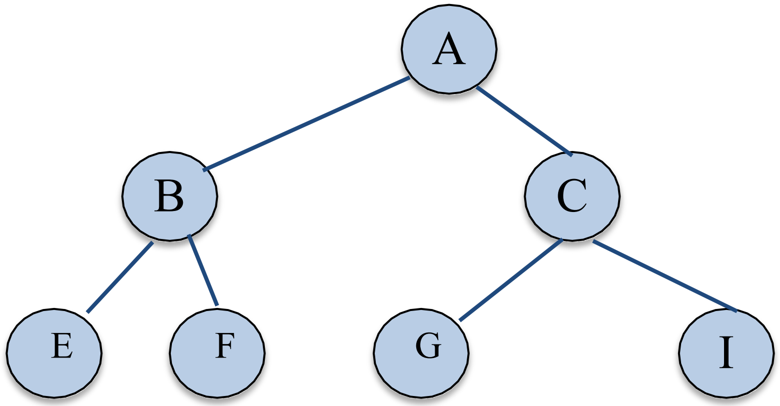


Tree ADT

The positions in a tree are its nodes.

- Accessor methods:
  - position root()
  - position parent(p)
  - PositionList children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - ObjectList elements()
  - PositionList positions()
  - swapElements(p, q)
  - object replaceElement(p, o)

Tree Traversal



A traversal visits the nodes of a tree in a systematic manner.

If it is binary tree, we have preorder, inorder and postorder traversal.

postorder: a node is visited after its descendants

inorder: a node is visited after its descendants

$O(n)$

Algorithm ***preOrder***(*v*)
If v is not null
visit(*v*)
preOrder (*v.left*)
preOrder (*v.right*)

Order: ABEFCGI

Algorithm ***inOrder***(*v*)
If v is not null
inOrder (*v.left*)
visit(*v*)
inOrder (*v.right*)

Order: EBFAGCI

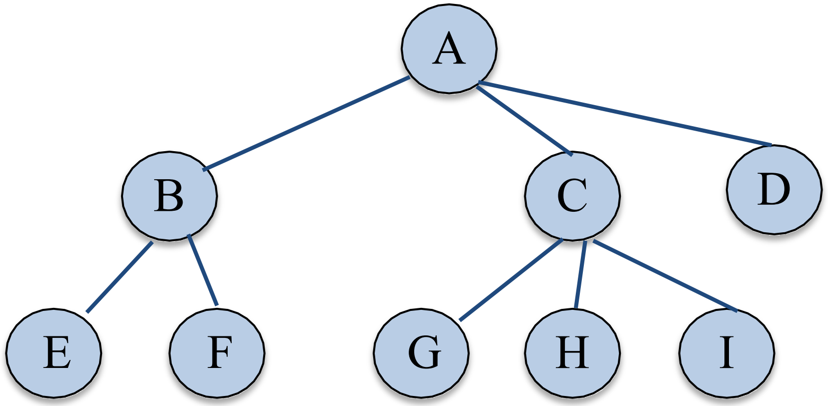
- postorder: a node is visited after its descendants

$O(n)$

Algorithm ***postOrder***(*v*)
If v is not null
postOrder (*v.left*)
postOrder (*v.right*)
visit(*v*)

Order: EFBGICA

Tree Traversal



A traversal visits the nodes of a tree in a systematic manner.

- preorder: a node is visited before its descendants

$O(n)$

```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder (w)
```

preOrder(A) visits ABEFCGHID

- postorder: a node is visited after its descendants

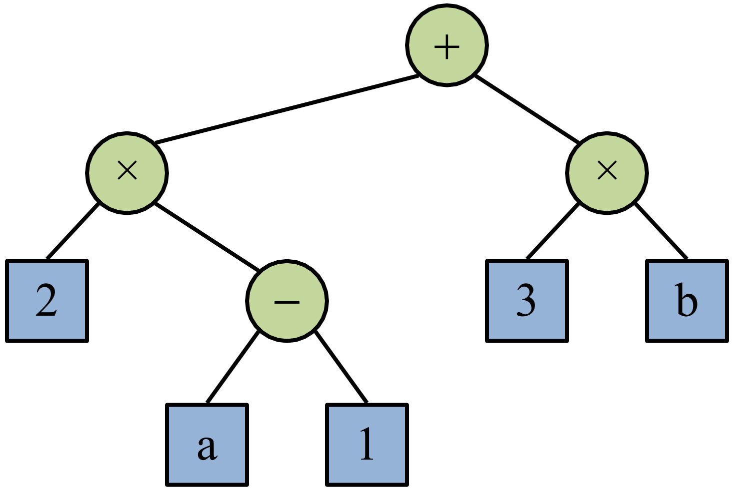
$O(n)$

```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```

postOrder(A) visits EFBGHICDA

Printing Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand/operator when visiting node
  - print "(" before visiting left
  - print ")" after visiting right



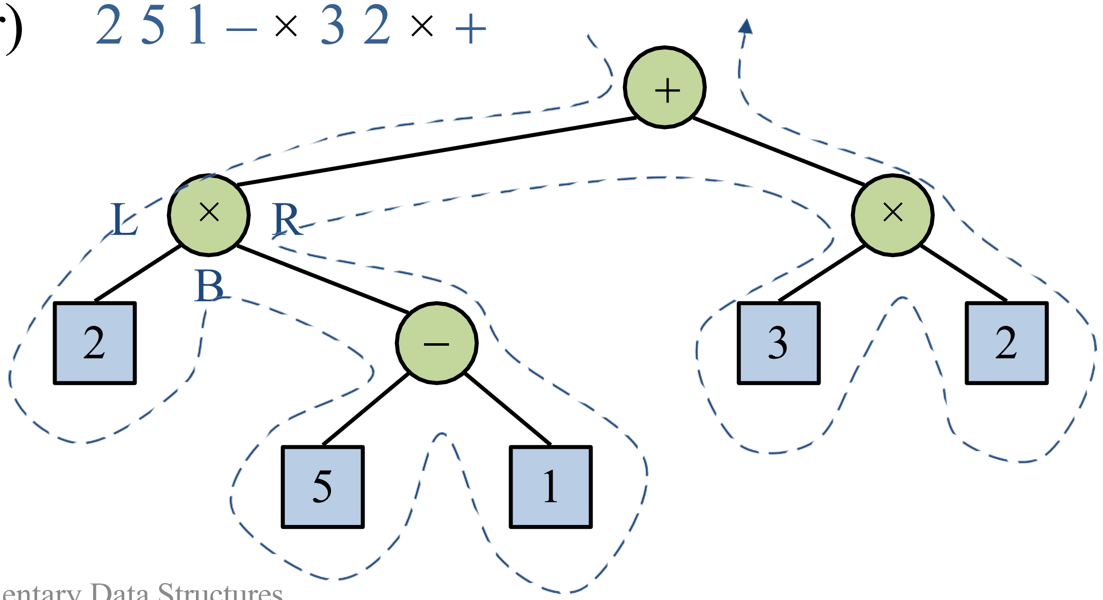
$O(n)$

Algorithm *inOrder_S*(*v*)
    if *isInternal* (*v*)
        print("(")
        inOrder_S(*v.left*)
    print(*v.element* ())
    if *isInternal* (*v*)
        inOrder_S (*v.right*)
        print (")")

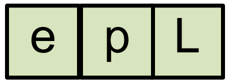
$((2 \times (a - 1)) + (3 \times b))$

Euler Tour Traversal

- Generic traversal of a binary tree
- Includes preorder, postorder, and inorder traversals as special cases
- Walk around the tree and visit each node three times:
  - on the left (preorder) $+ \times 2 - 5\ 1 \times 3\ 2$
  - from below (inorder) $2 \times 5 - 1 + 3 \times 2$
  - on the right (postorder) $2\ 5\ 1 - \times\ 3\ 2 \times +$

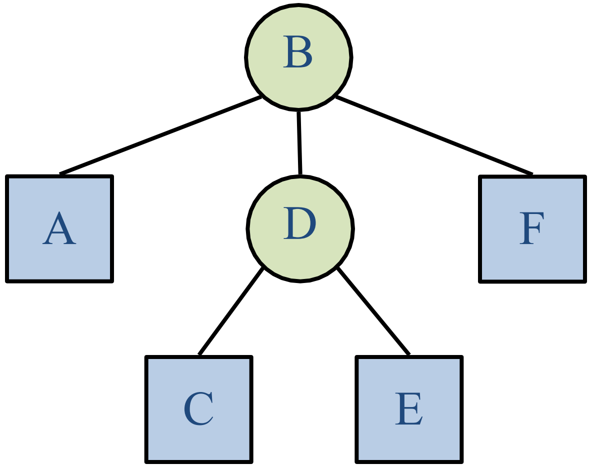


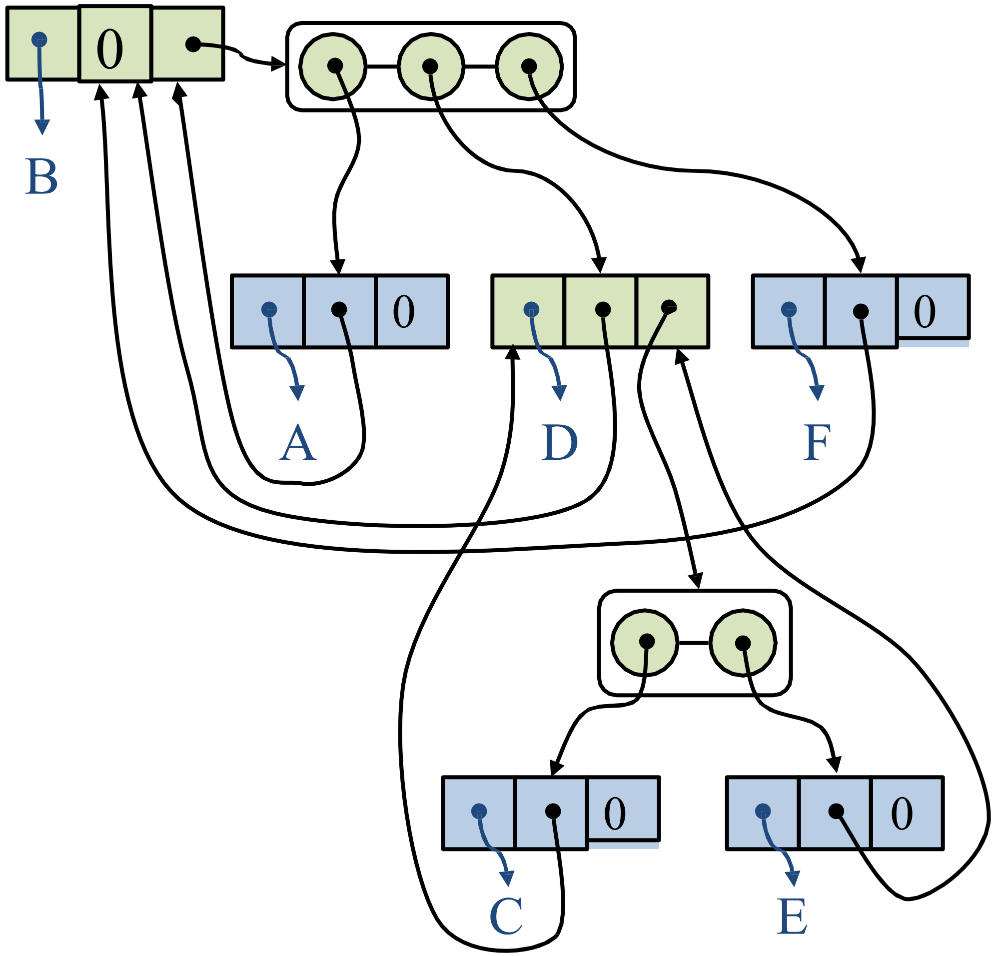
Linked Data Structure for Representing Trees



A node stores:

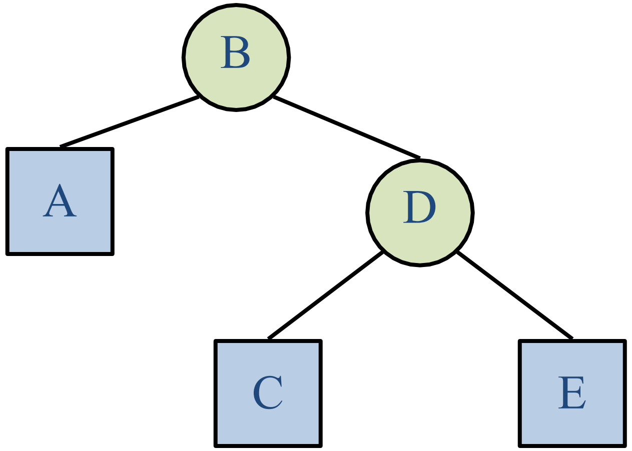
- Element (e)
- parent node (p)
- sequence of children nodes(L)

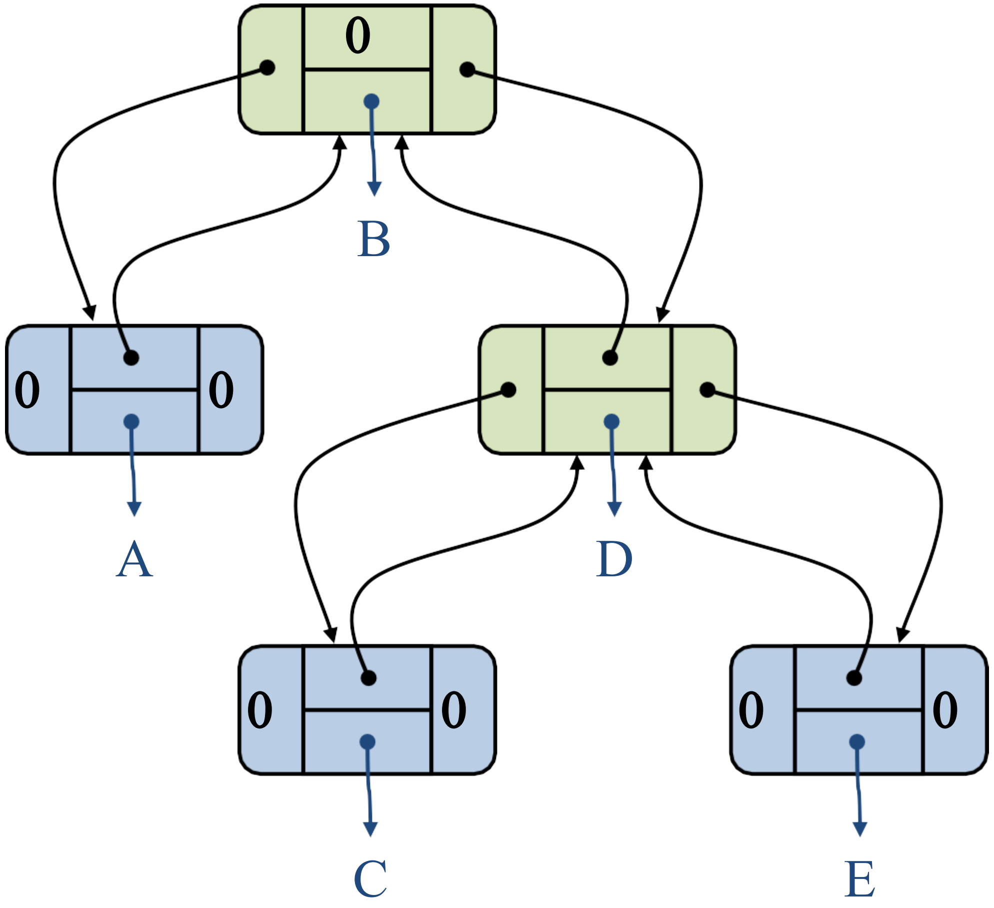




Linked Data Structure for Binary Trees

A node stores:
- element
- parent node
- left node
- right node





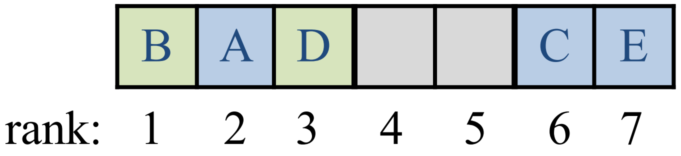
Array-Based Representation of Binary Trees

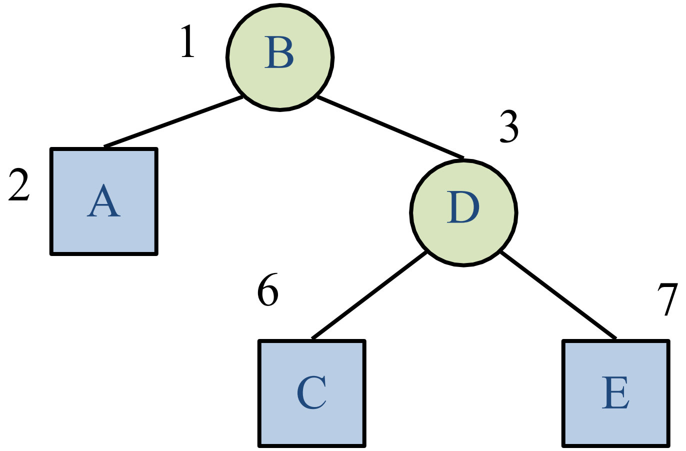
Nodes are stored in an array

- *rank*(root) = 1
- If *rank*(node) = *i*, then

  rank(leftChild) = 2**i*

  rank(rightChild) = 2**i* + 1





Ex: ‘A’ is left child of B

rank(A) = 2 * *rank*(B)

= 2 * 1 = 1

Ex: ‘E’ is right child of D

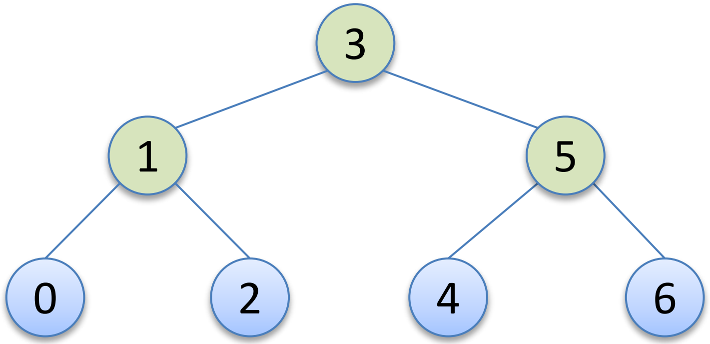
rank(E) = 2 * *rank*(D) + 1

= 2 * 3 + 1

= 7

Perfect, and Complete Binary Trees (cont.)

A *perfect binary tree* is a full binary tree of height n with exactly
$2^n - 1$ nodes
In this case, $n = 3$ and $2^n - 1 = 7$

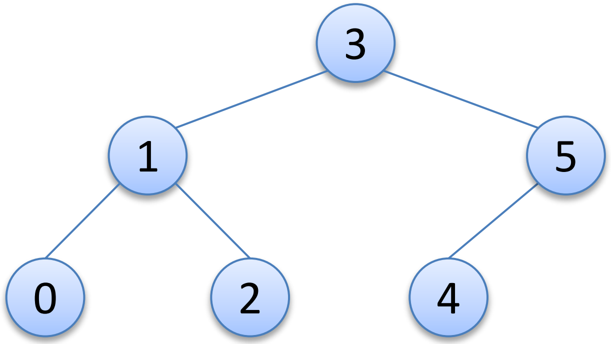


Height h = O(logn)

	3	1	5	0	2	4	6
rank:	1	2	3	4	5	6	7

Perfect, and Complete Binary Trees (cont.)

A *complete binary tree* is a perfect binary tree through level n - 1 with some extra leaf nodes at level n (the tree height), all toward the left



Height h = O(logn)

	3	1	5	0	2	4	
rank:	1	2	3	4	5	6	7

END