Elementary Data Structures
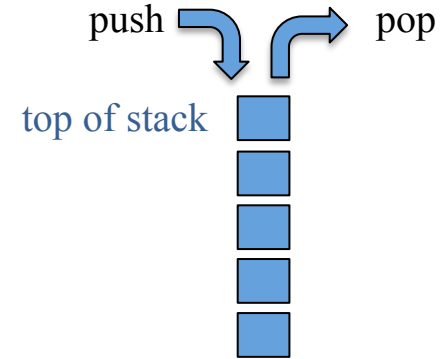
Stacks & Queues
Lists, Vectors, Sequences
Amortized Analysis
Trees

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

- 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 $t+1$

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

$O(1)$

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

$O(1)$

S

0 1 2

...

t

Elementary Data Structures

4

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 $t = N\text{-}1$ **then**
    $A \leftarrow$ new array of size $N*$
    for $i \leftarrow 0$ **to** t **do**
      $A[i] \leftarrow S[i]$
      $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

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

- 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 $k = n/c$ times

- Total time $T(n)$ of a series of n push operations is proportional to:
$$n + c + 2c + 3c + 4c + \ldots + kc$$
$$= n + c(1 + 2 + 3 + \ldots + k)$$
$$= n + ck(k + 1)/2$$

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

- The amortized time of a push operation is $O(n)$

Doubling Strategy

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

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$

$$= n + 2^{k+1} - 1$$

$$= n + 2^{\log n + 1} - 1$$

$$= n + 2^{\log n}\, 2^1 - 1$$

$$= n + 2n - 1$$

$$= 3n - 1$$

> Recall the summation of this geometric series:
>
> $$2^0 + 2^1 + \ldots + 2^k = 2^{k+1} - 1$$

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

Accounting Method Analysis

- The accounting method determines amortized running time using a scheme of credits and debits

- View computer as a coin-operated devices that needs $1 (cyber-dollar) for each primitive operation
  - Set up an amortization scheme for charging operations
  - Must always have enough money to pay for actual cost of operation
  - Total cost of the series of operations is no more than the total amount charged

- (amortized time) ≤ (total $ charged) / (# operations)

Accounting Method Analysis: Doubling Strategy

- How much to charge for a push operation?
  - Charge $1? No, not enough $$ to copy old elements
  - Charge $2? No, not enough $$ to copy old elements a second time
  - Charge $3 for a push: use $1 to pay for push, save $2 to pay for copying all old elements into new array.
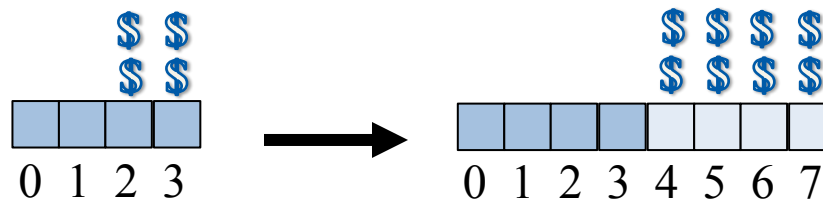
- Each push runs in O(1) amortized time; n pushes run in O(n) time.

Queue ADT

enqueue ⟶ ▢ ▢ ▢ ▢ ⟶ dequeue

end   front
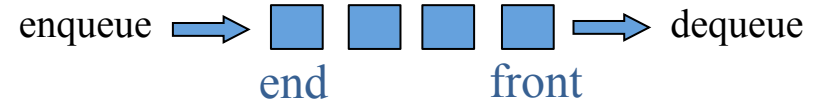
- Container that stores arbitrary objects

- 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
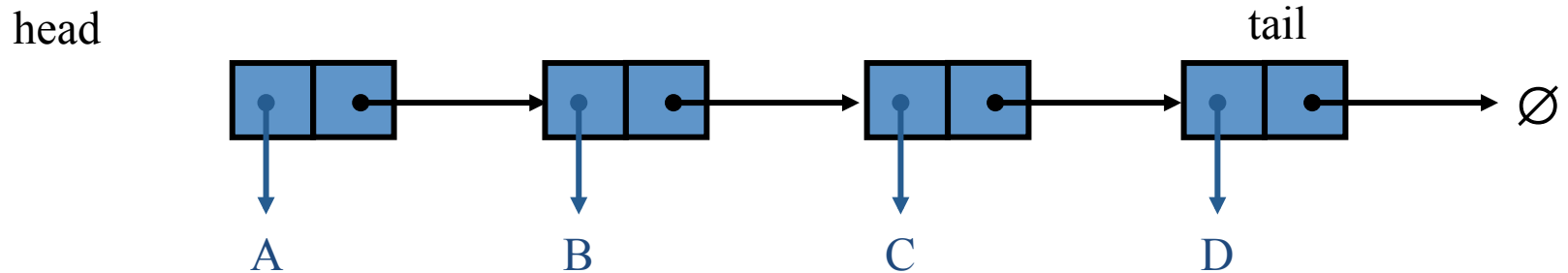
  – boolean isEmpty(): returns whether no elements are stored

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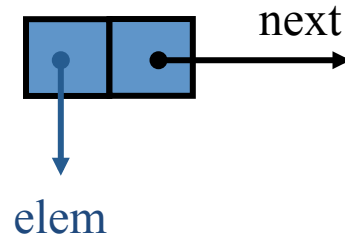
- Each node stores an element and a link to the next node

Queue with a Singly Linked List

- Singly Linked List implementation
 - front is stored at the first node
  - end is stored at the last node
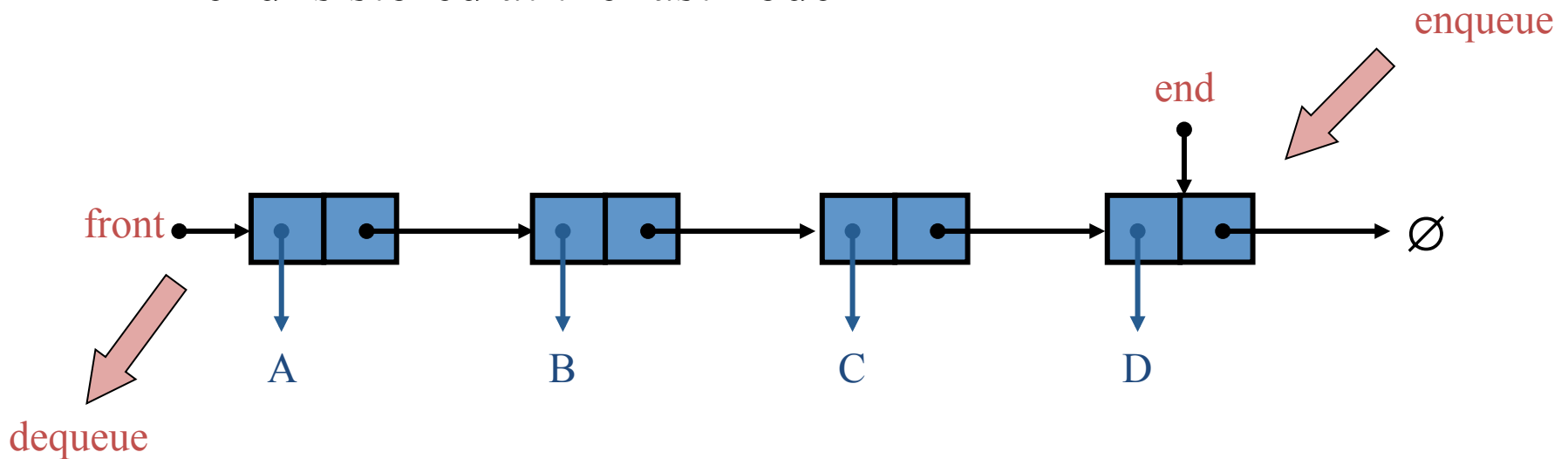
- Space used is $O(n)$ and each operation takes $O(1)$ time

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
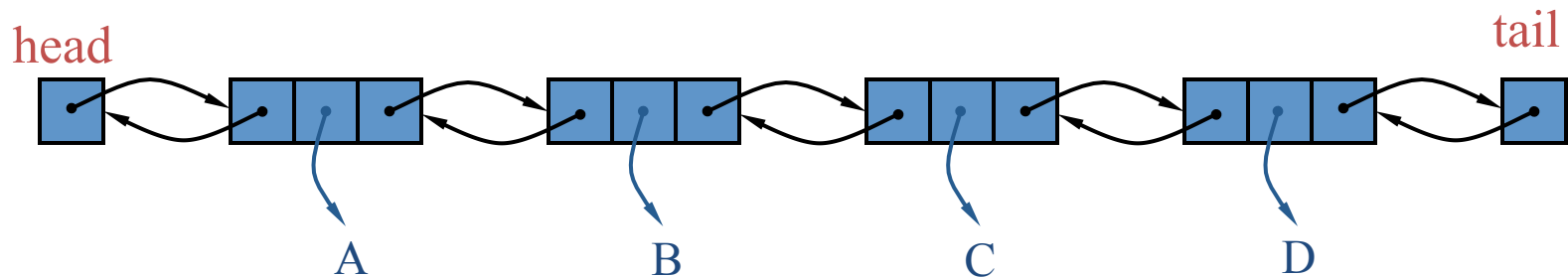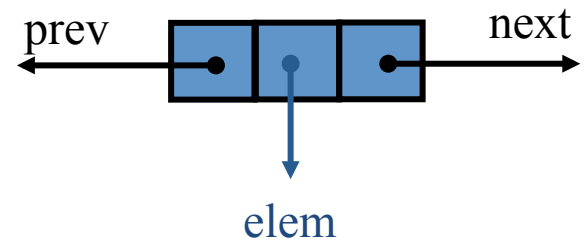
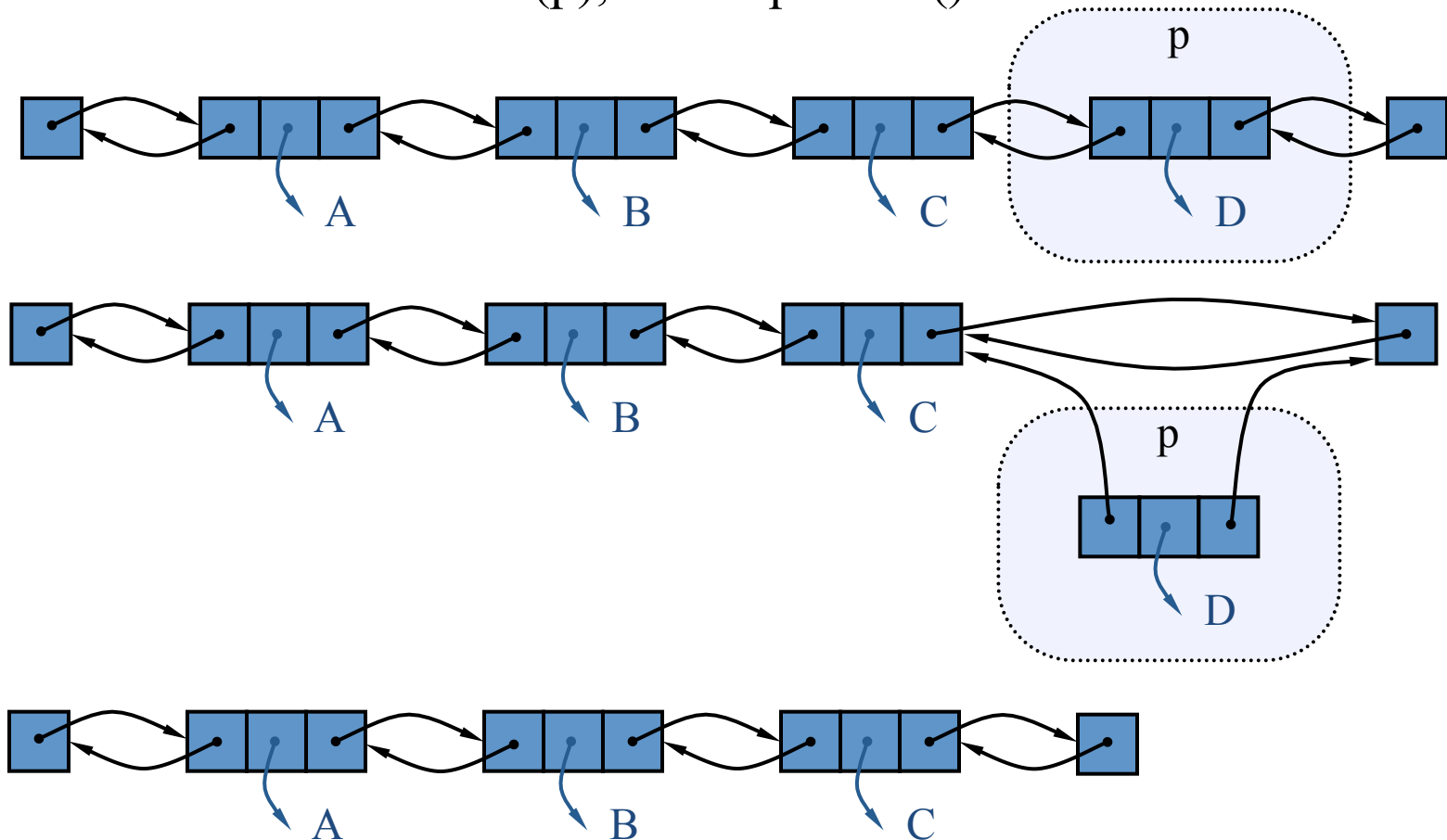# Insertion: insertAfter(*p*, *X*)
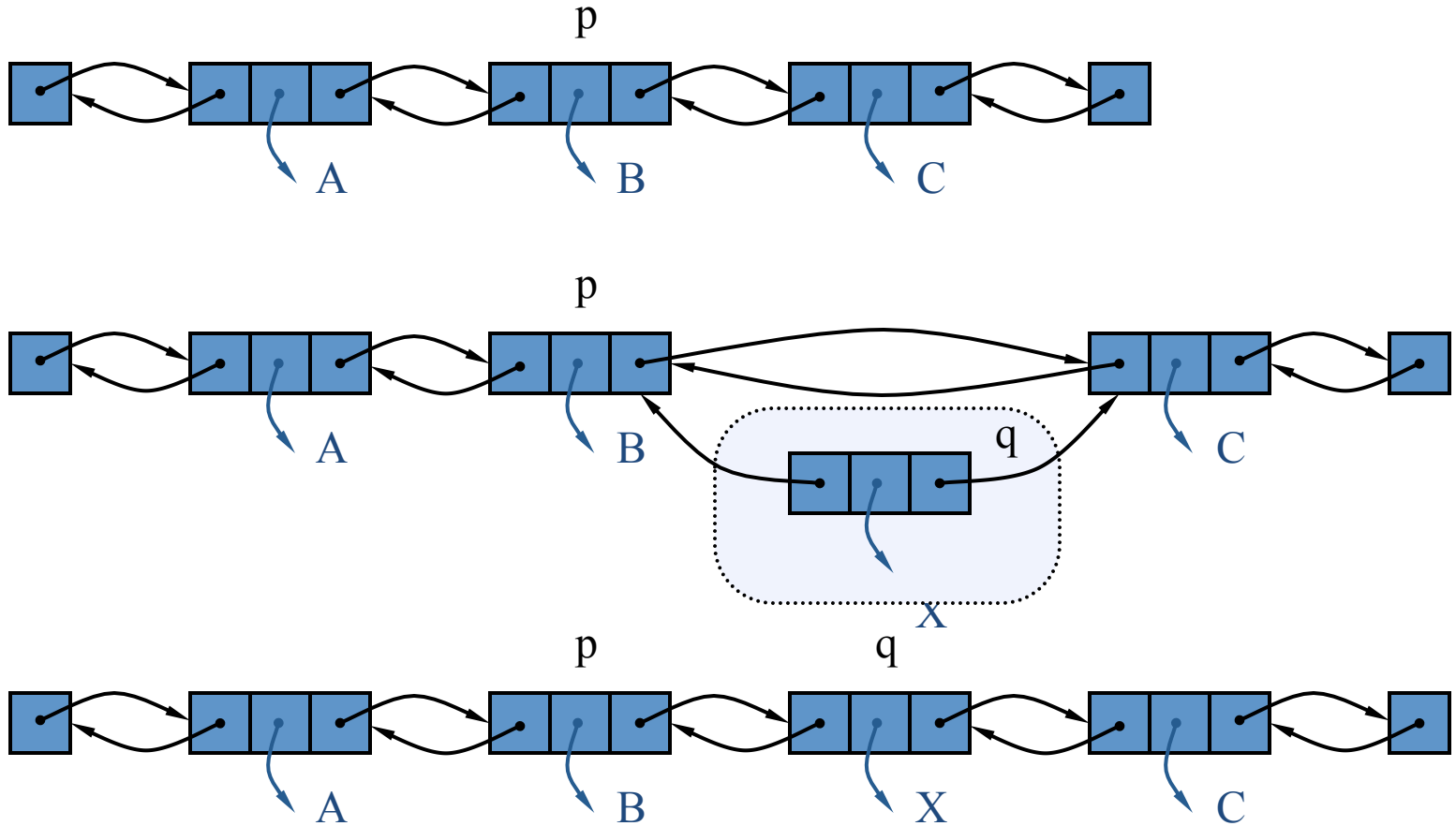
Deletion: remove(*p*)

- 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)
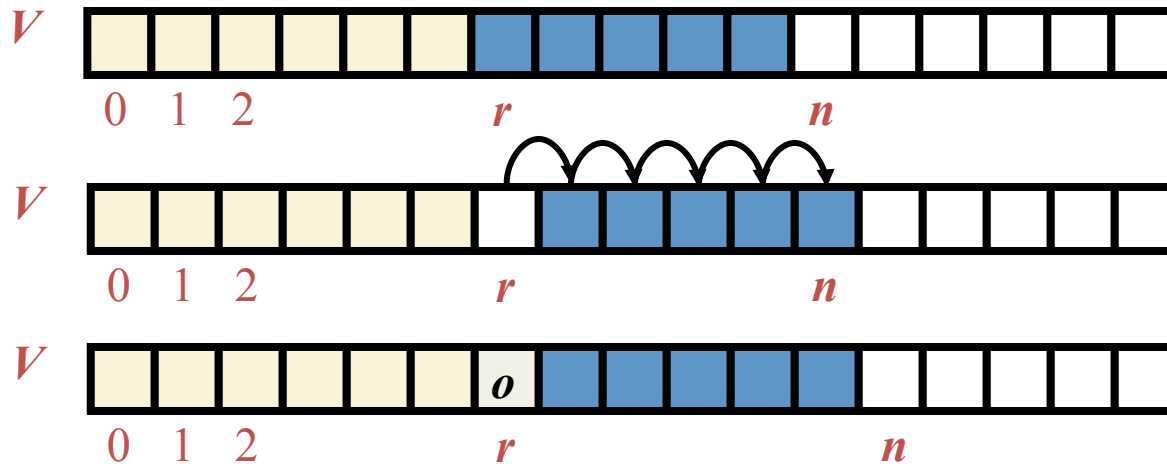
Array-based Vector

- 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*]

V

0 1 2   *r*   *n*
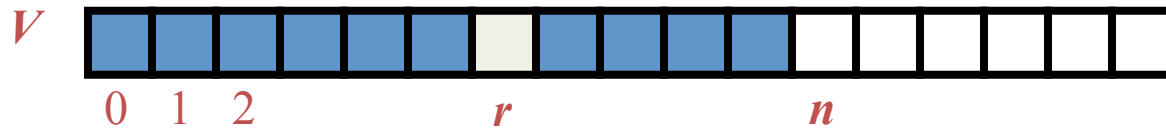
Insertion: insertAtRank(*r, o*)

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

- 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 $n - r - 1$ elements $V[r + 1], \ldots, V[n - 1]$

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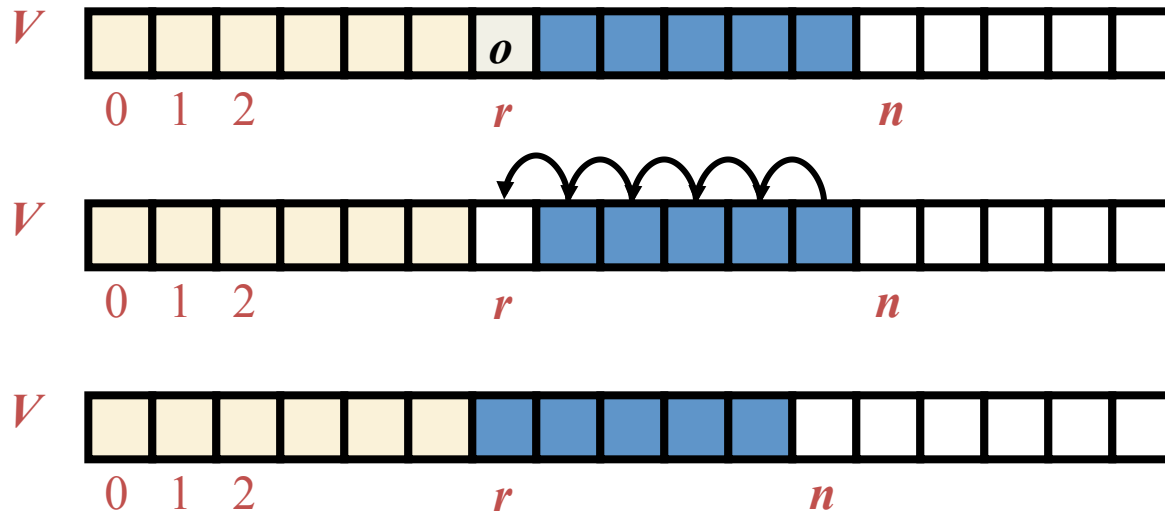
Sequence

- 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	$\mathbf{O(1)}$	$O(n)$
first, last, before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$\mathbf{O(1)}$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$\mathbf{O(1)}$
remove	$O(n)$	$\mathbf{O(1)}$

Tree

- Stores elements hierarchically

- Each node has a parent-child relation

- Direct applications:
 - Organizational charts
  - File systems
  - Programming environments
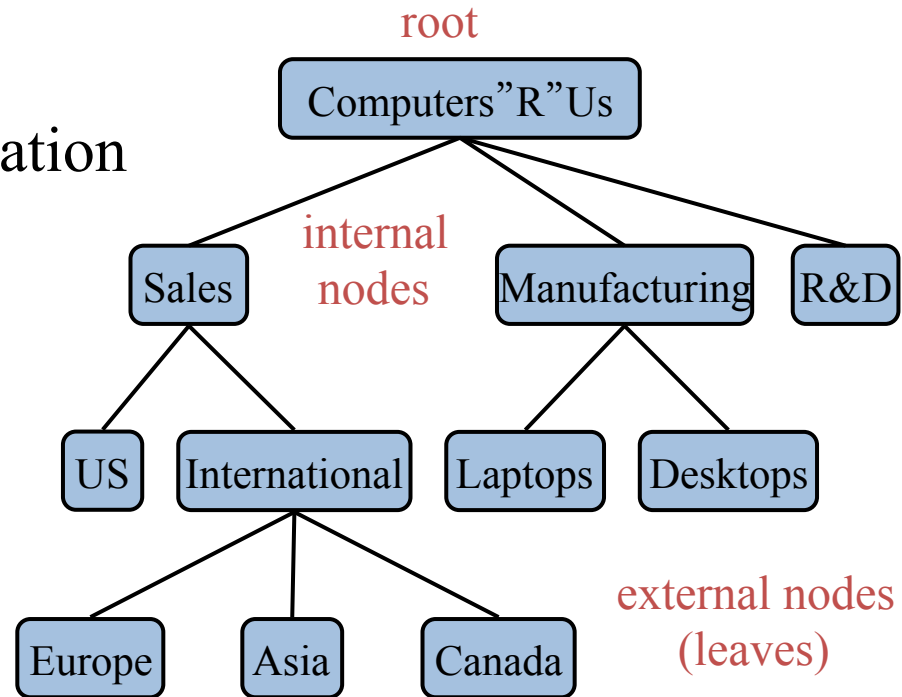
Tree ADT

The positions in a tree are its nodes.

- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
  - positionIterator positions()
  - swapElements(p, q)
  - object replaceElement(p, o)

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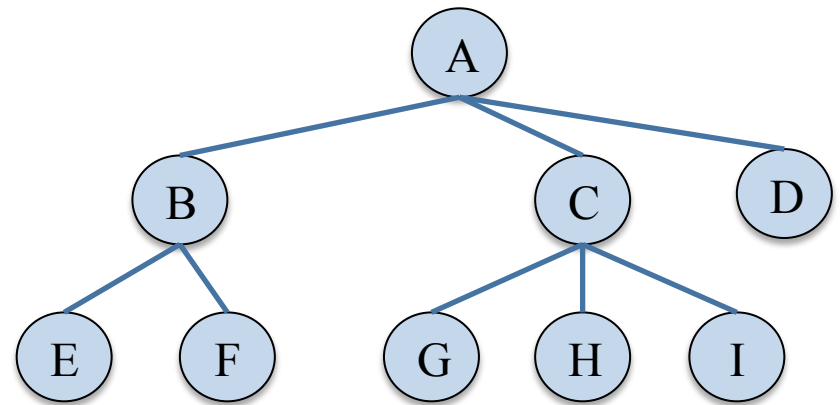
A traversal visits the nodes of a tree in a systematic manner.

- 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

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 right)

- 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
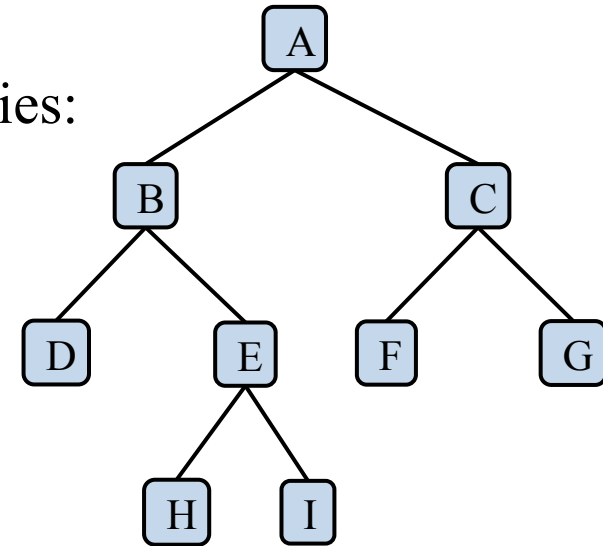
Arithmetic Expression Tree

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

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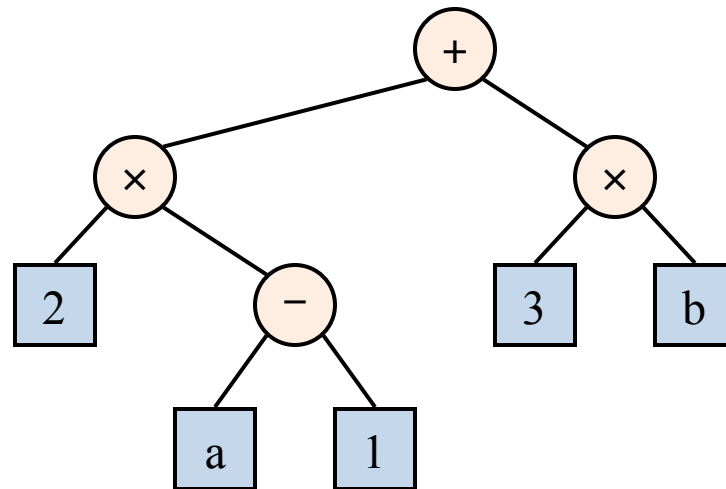
Decision Tree

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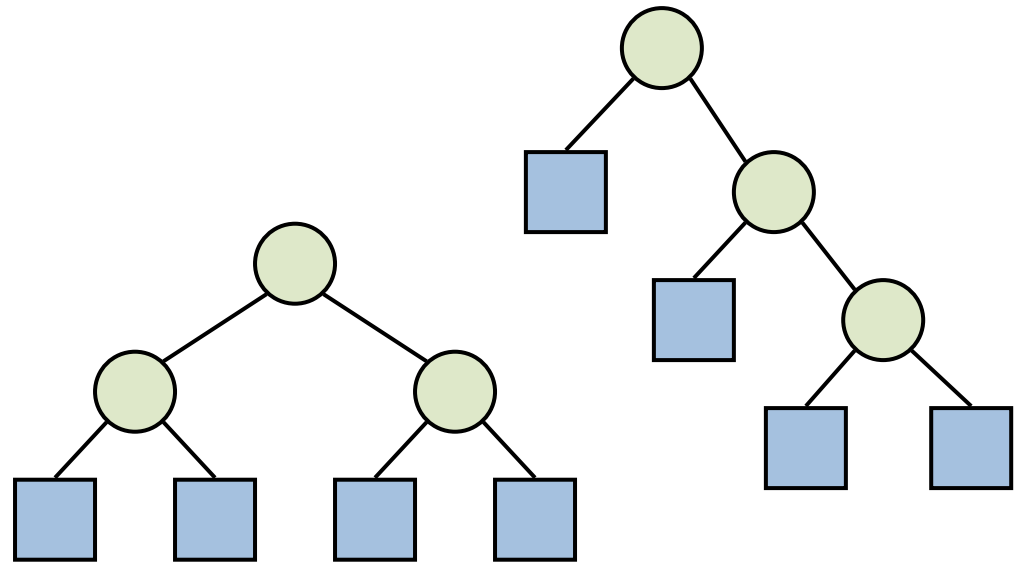- Ex: dining decision
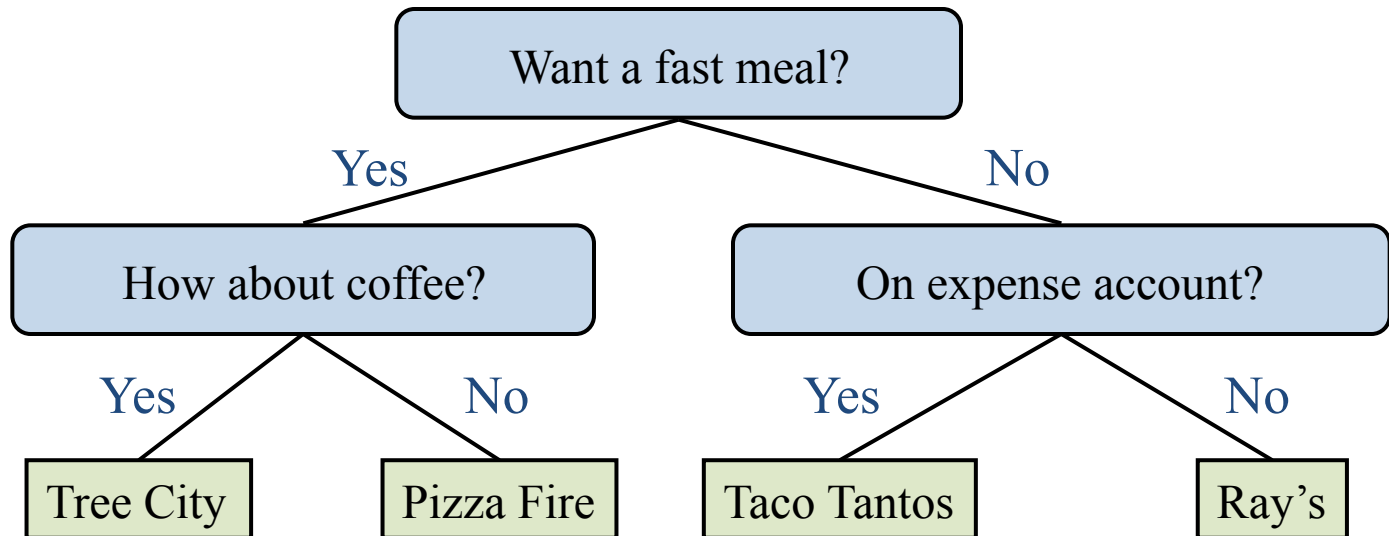
Properties of Binary Trees

n number of nodes

e number of external nodes

i number of internal nodes

h height (max distance from root)

Properties:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
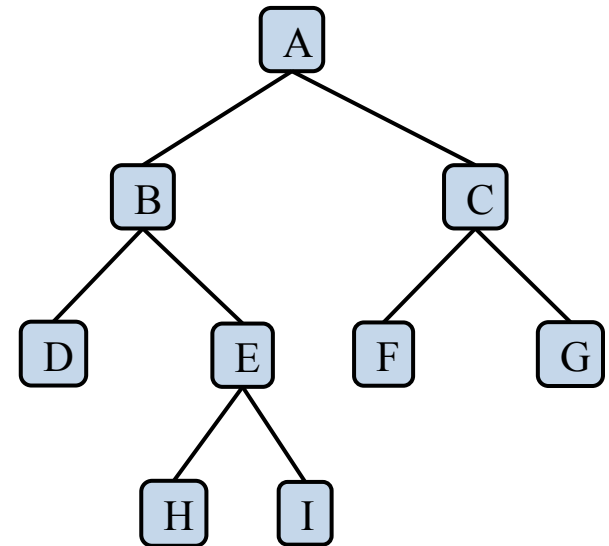- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$

Inorder Traversal of a Binary Tree

- inorder traversal: visit a node after its left subtree and before its right subtree

$O(n)$

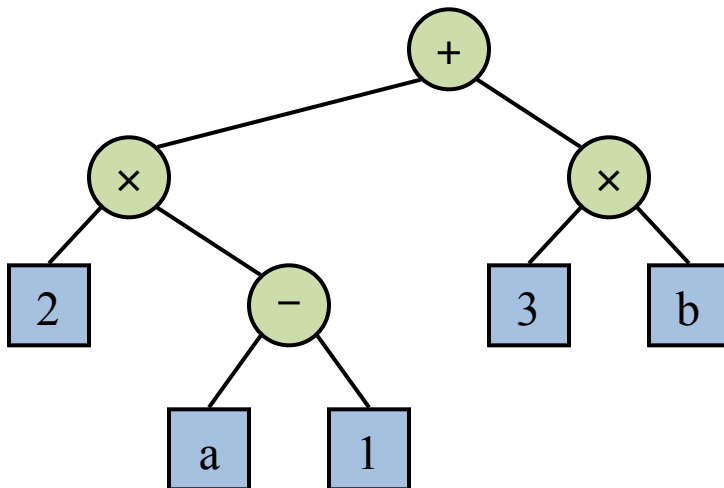> **Algorithm** *inOrder*(*v*)
>   **if** *isInternal* (*v*)
>   *inOrder* (*leftChild* (*v*))
>   *visit*(*v*)
>   **if** *isInternal* (*v*)
>   *inOrder* (*rightChild* (*v*))

Ex: DBHEIAFCG

Printing Arithmetic Expressions

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

$O(n)$

Algorithm *printExpression(v)*
    if *isInternal (v)*
        print("(")
        inOrder (leftChild (v))
    print(v.element ())
    if *isInternal (v)*
        inOrder (rightChild (v))
        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) $2\ 5\ 1 - \times 3\ 2 \times +$
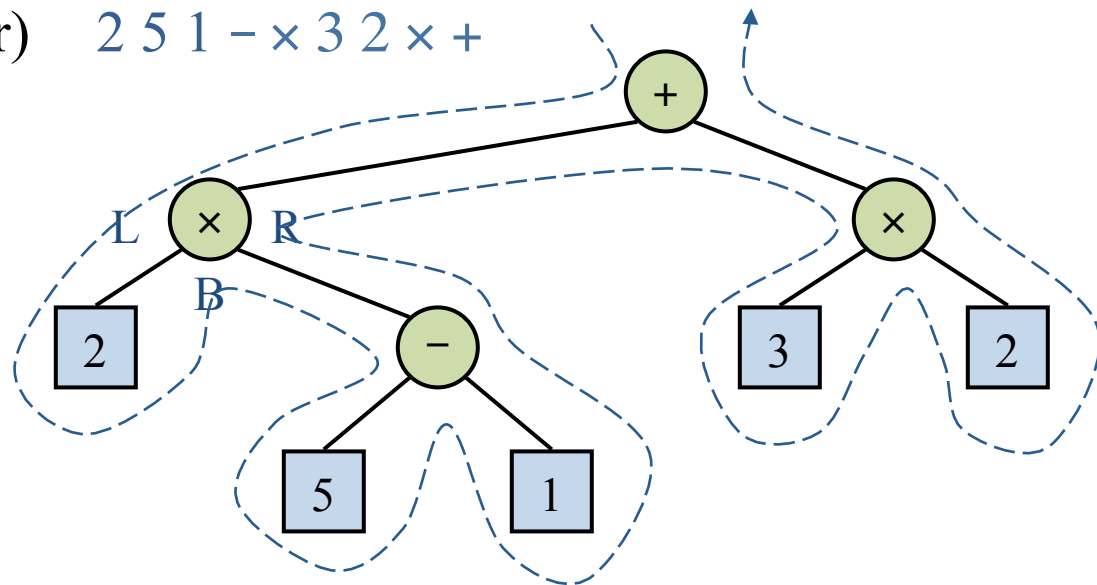
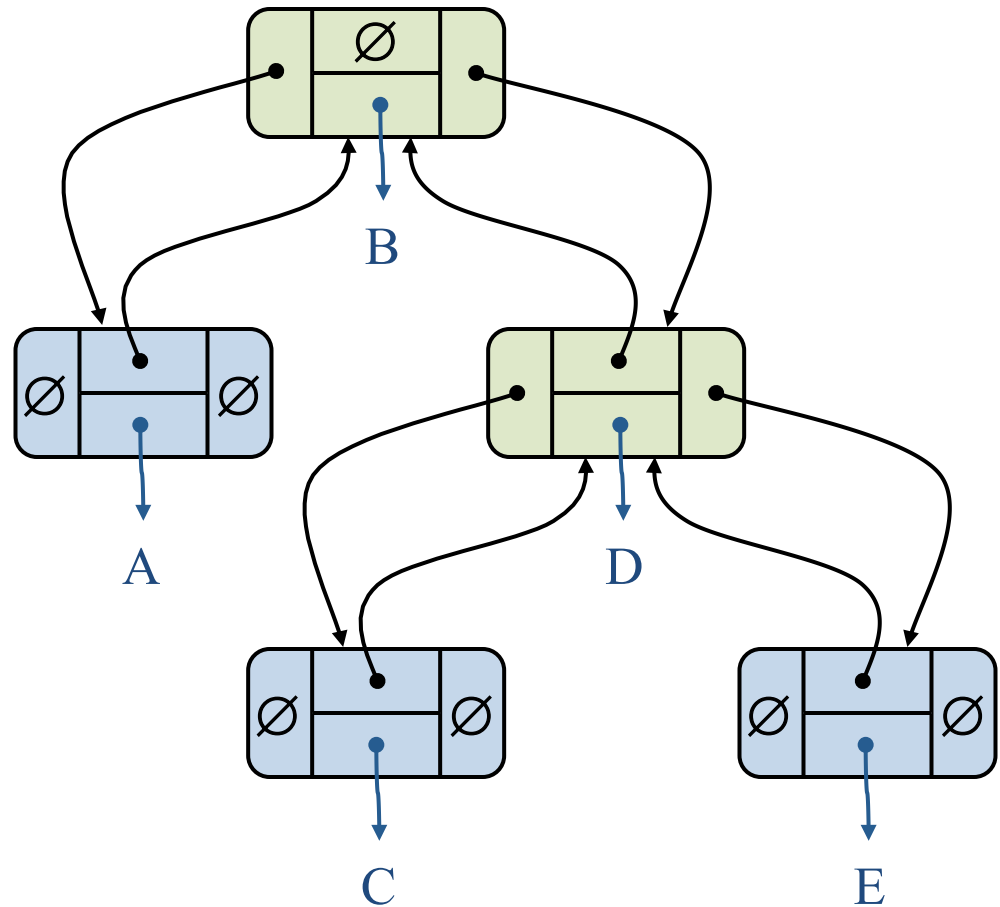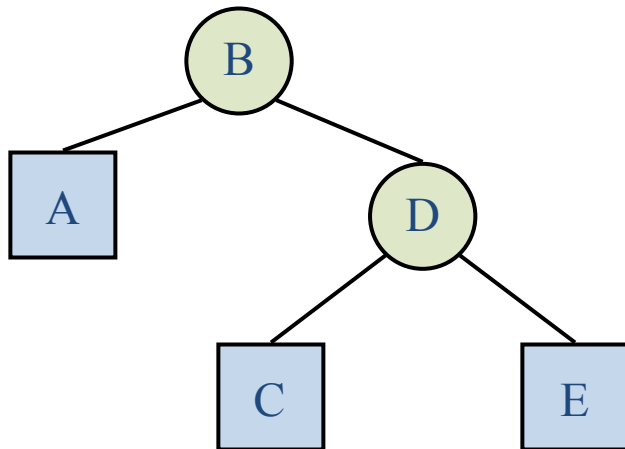# Linked Data Structure for Representing Trees

A node stores:

- element

- parent node

- sequence of children nodes
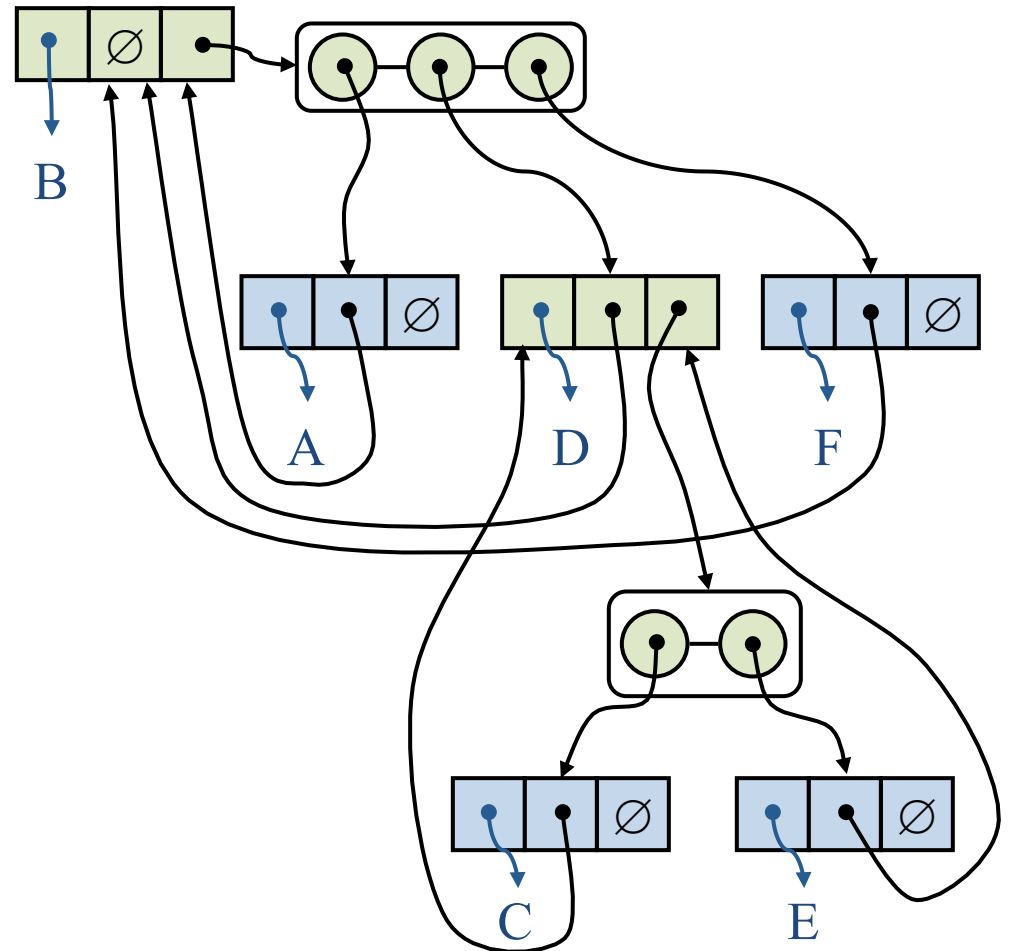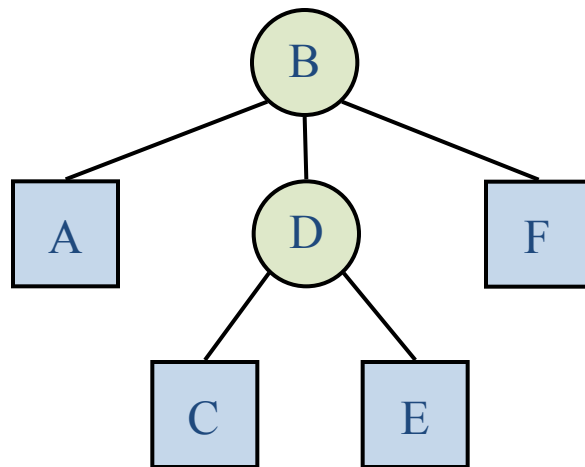
Linked Data Structure for Binary Trees
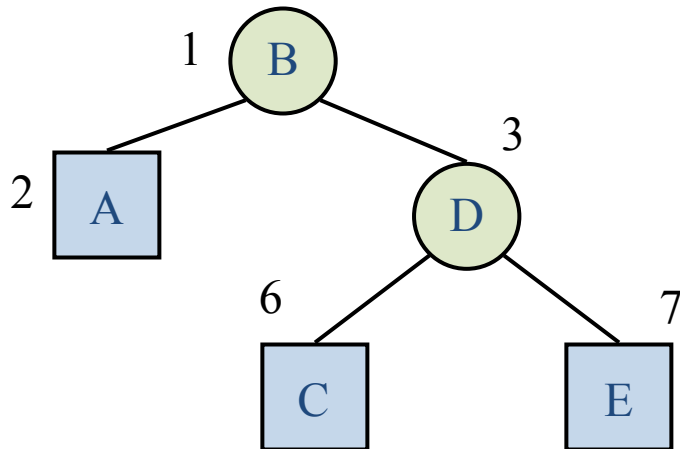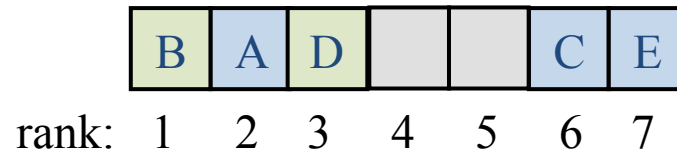
A node stores:

- element
- parent node
- left node
- right node

Array-Based Representation of Binary Trees

Nodes are stored in an array

- $rank(\text{root}) = 1$
- If $rank(\text{node}) = i$, then
  $rank(\text{leftChild}) = 2*i$
  $rank(\text{rightChild}) = 2*i + 1$

B	A	D			C	E

rank: 1   2   3   4   5   6   7

Ex: 'A' is left child of B
$rank(A) = 2 * rank(B)$
$\qquad\quad = 2 * 1 = 1$

Ex: 'E' is right child of D
$rank(E) = 2 * rank(D) + 1$
$\qquad\quad = 2 * 3 + 1$
$\qquad\quad = 7$