Elementary Data Structures

Stacks & Queues

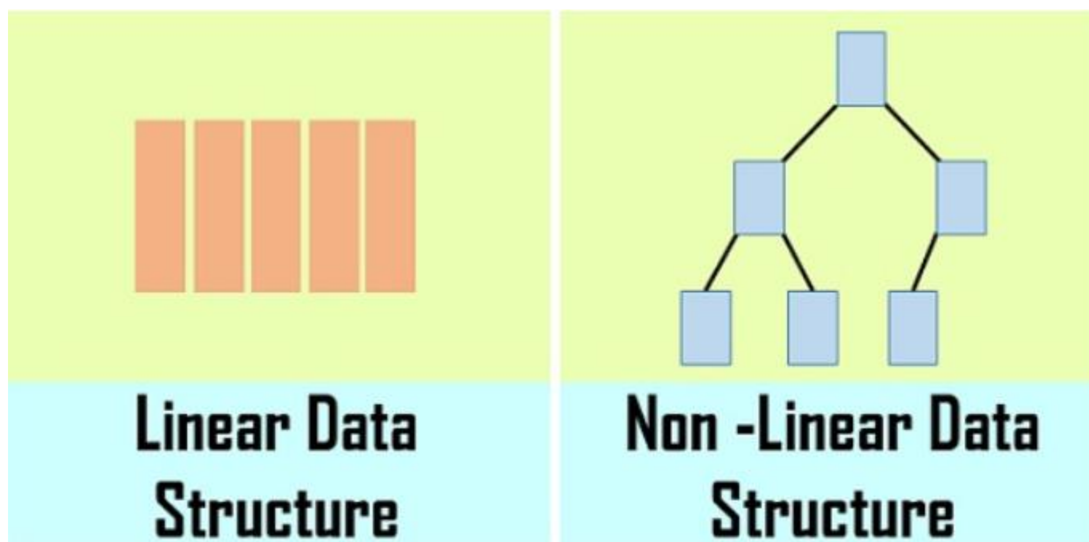
Lists, Vectors, Sequences

Amortized Analysis

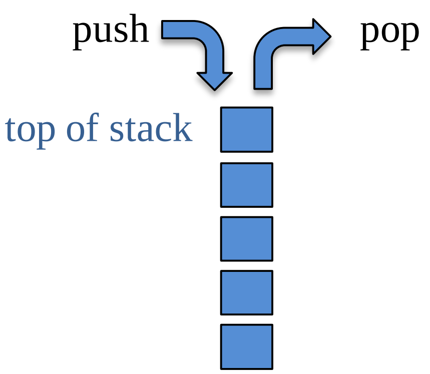
Trees

Elementary Data Structures

- Linear Data Structures:
  - ➢ Stacks, Queues, Vectors, Lists and Sequences
- Hierarchical Data Structures (non-linear):
  - ➢ Tree



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 $t+1$

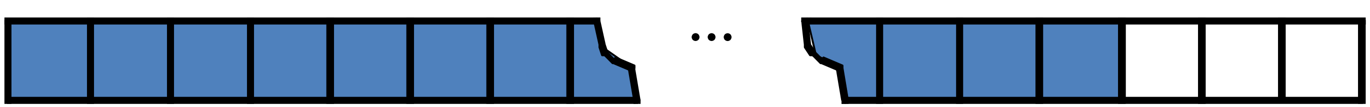
Algorithm ***push***(o):
  if $t = N-1$ **then**
    throw ***FullStackException***
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

$O(1)$

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

$O(1)$

S



0 1 2 … t

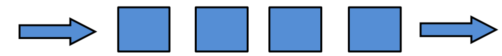
Amortization

- Amortization: analysis tool to understand running times of algorithms that have steps with widely varying performance.
- In an ***amortized analysis***, we average the running time T(n) required to perform a sequence of data-structure operations over all the operations performed, i.e., $\boldsymbol{T(n) / n}$
- Amortization takes into an account the interactions between the operations rather than focusing ton each operations separately.

Amortization

- Let us have another operation in Stack.
  - clearStack():Remove all elements of stack.
  - The running time of clearStack() is $\theta(n)$.
- Consider a series of n operations on empty stack.
  - What is the running of clearStack() on these n operations?
  - There might be as many as $O(n)$ clear operations in this series, so we may say that the running of this series is $O(n^2)$.
  - This is true but an overstatement.
  - Since there is an interaction between these operations, the amortizations analysis can show that the running of the entire series of n operations is O(n).
  - So the average running time of any operation is O(n).

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

Algorithm dequeue():

    if isEmpty() **then**

        throw a QueueEmptyException

    $temp \leftarrow Q[f]$

    $Q[f] \leftarrow$ **null**

    $f \leftarrow (f + 1) \bmod N$

    return $temp$

Algorithm enqueue(o):

    if size() $= N - 1$ **then**

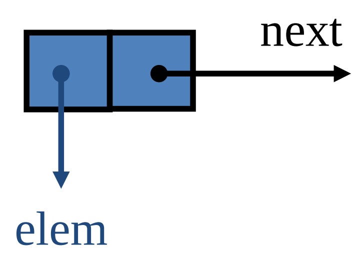
        throw a QueueFullException

    $Q[r] \leftarrow o$

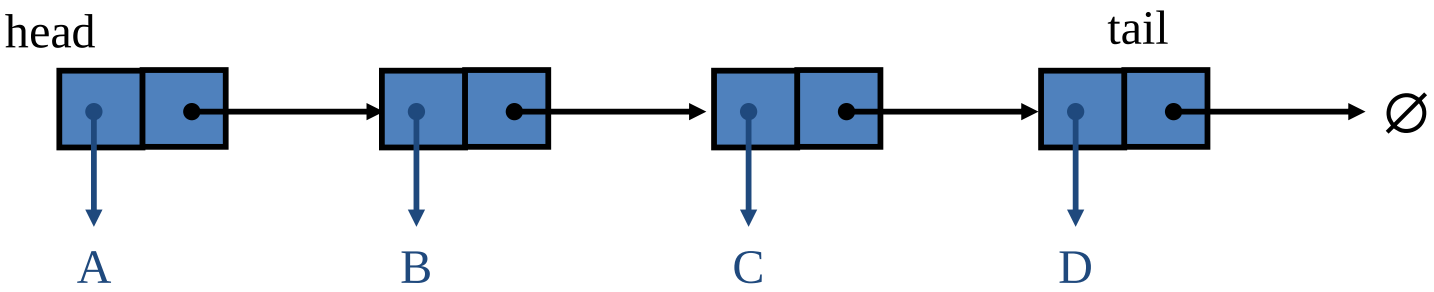
    $r \leftarrow (r + 1) \bmod N$

Singly Linked List

- A data structure consisting of a sequence of nodes

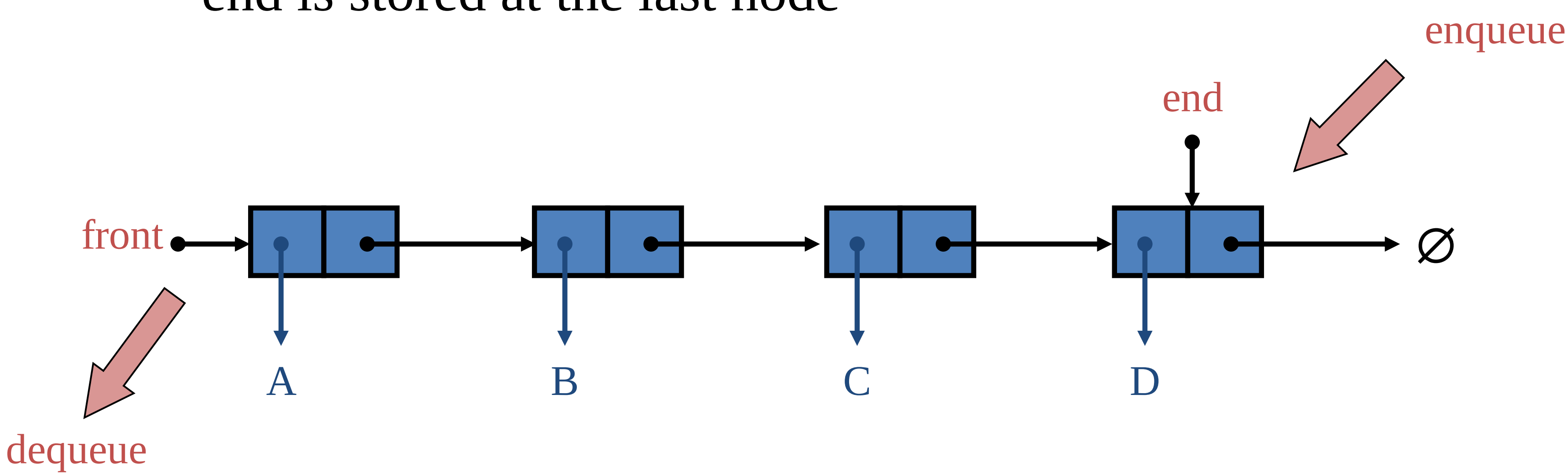


- Each node stores an element and a link to the next node



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

Vectors, Lists and Sequences

- Stacks and queues store elements according to a linear sequence determined by update operations that act on the "ends" of the sequence.
- Vectors, Lists and Sequences maintain linear orders while allowing for accesses and updates in the "middle."

Vector ADT

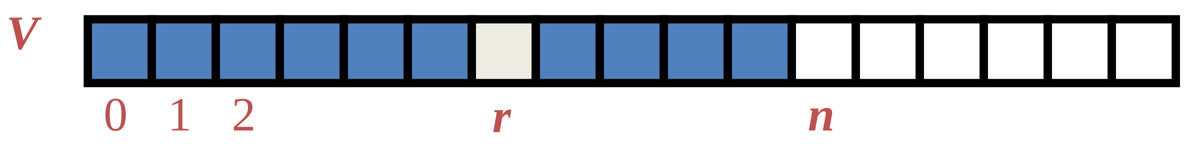
- A linear sequence that supports access to its elements by their **rank** (number of elements preceding it).

- **Rank** is similar to an array index.
  - but we do not insist that an array should be used to implement a sequence in such a way that the element at rank 0 is stored at index 0 in the array.
- The rank of an element may change whenever the sequence is updated.

- Main operations:
  - size() O(1)
  - isEmpty() O(1)
  - elemAtRank(r) O(1)
  - replaceAtRank(r, e) O(1)
  - insertAtRank(r, e) O(n)
  - removeAtRank(r) O(n)

Array-based Vector

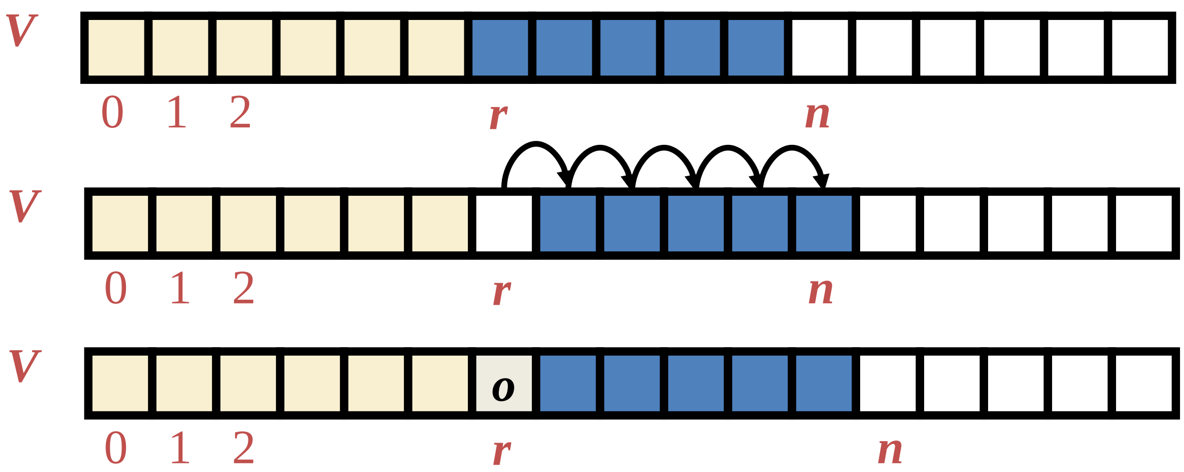
- Use an array ***V*** of size ***N***
- A variable ***n*** keeps track of the size of the vector (number of elements stored)
- ***elemAtRank***(***r***) is implemented in ***O***(1) time by returning ***V***[***r***]



Insertion: insertAtRank(*r*, *o*)

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

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

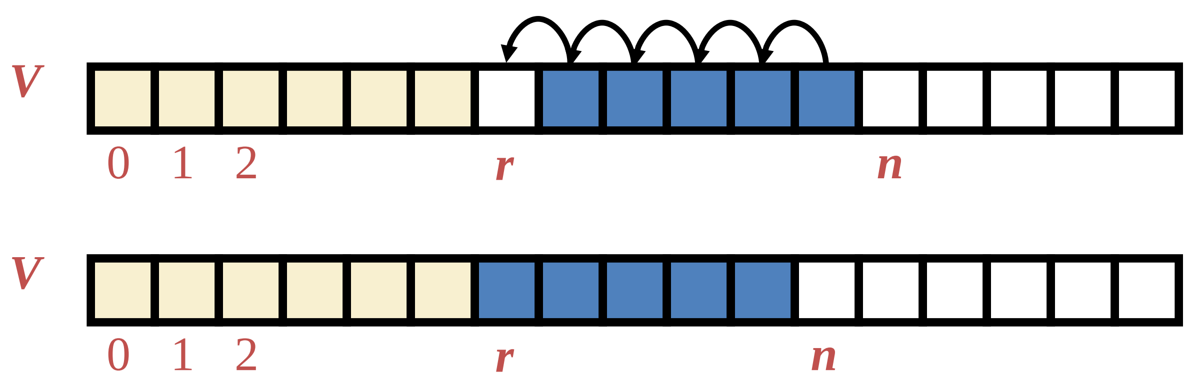


- We could use an extendable array when more space is required

Deletion: removeAtRank(r)

- Need to fill the hole left by the removed element by shifting backward the $\boldsymbol{n} - \boldsymbol{r} - 1$ 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



Algorithm insertAtRank(r, e):

  for $i = n-1, n-2, \ldots, r$ **do**

    $A[i+1] \leftarrow A[i]$ {make room for the new element}

  $A[r] \leftarrow e$

  $n \leftarrow n+1$

Algorithm removeAtRank(r):

  $e \leftarrow A[r]$ {e is a temporary variable}

  for $i = r, r+1, \ldots, n-2$ **do**

    $A[i] \leftarrow A[i+1]$ {fill in for the removed element}

  $n \leftarrow n-1$

  return e

Algorithm 2.6: Methods in an array implementation of the vector ADT.

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)

List ADT

first(): Return the position of the first element of S; an error occurs if S is empty.

last(): Return the position of the last element of S; an error occurs if S is empty.

isFirst(p): Return a Boolean value indicating whether the given position is the first one in the list.

isLast(p): Return a Boolean value indicating whether the given position is the last one in the list.

before(p): Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.

after(p): Return the position of the element of S following the one at position p; an error occurs if p is the last position.

List ADT

replaceElement(p,e): Replace the element at position p with e, returning the element formerly at position p.

swapElements(p,q): Swap the elements stored at positions p and q, so that the element that is at position p moves to position q and the element that is at position q moves to position p.

insertFirst(e): Insert a new element e into S as the first element.

insertLast(e): Insert a new element e into S as the last element.

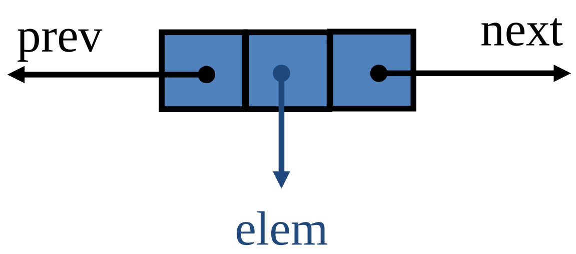
insertBefore(p,e): Insert a new element e into S before position p in S; an error occurs if p is the first position.

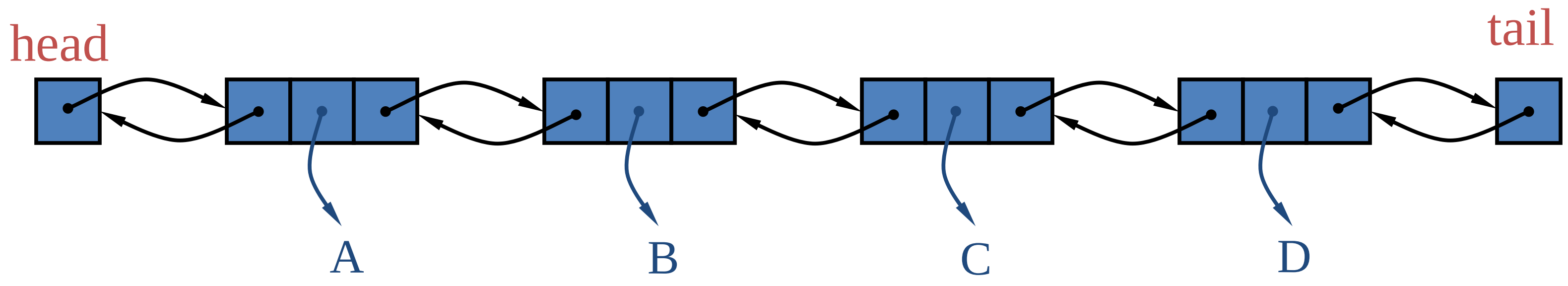
insertAfter(p,e): Insert a new element e into S after position p in S; an error occurs if p is the last position.

remove(p): Remove from S the element at position p.

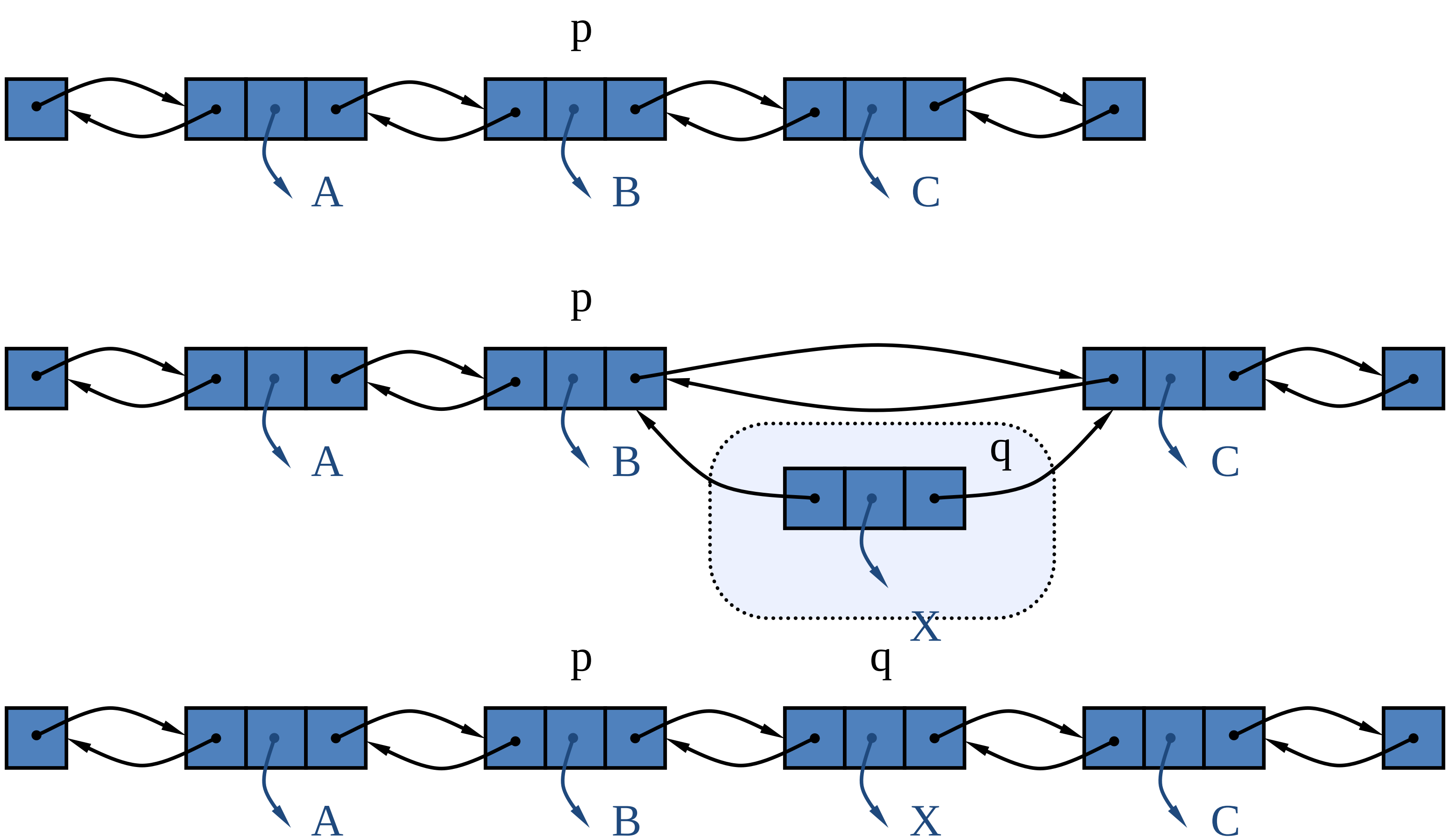
Doubly Linked List

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



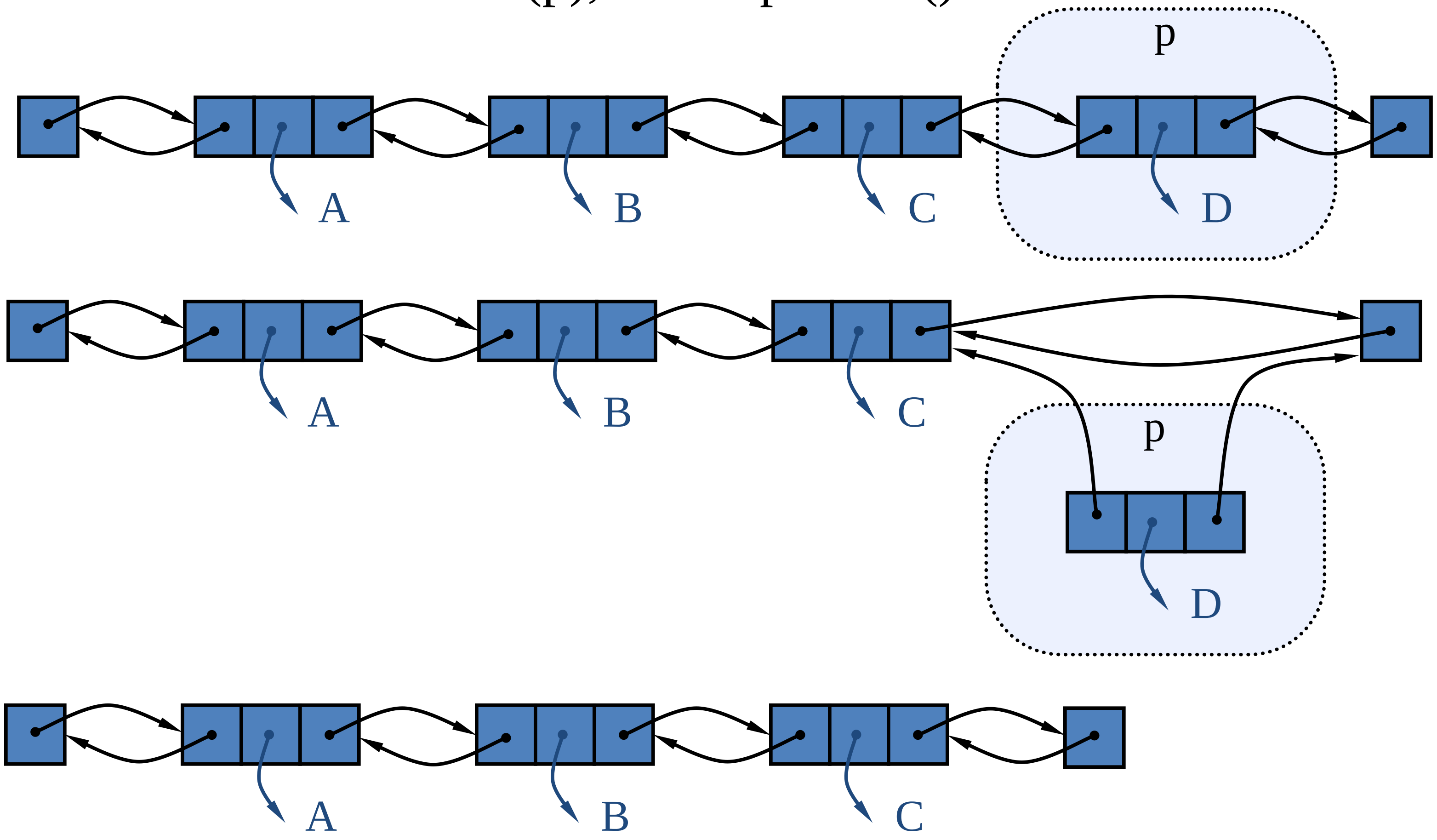


Insertion: insertAfter(*p*, *X*)



Deletion: remove(p)

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



Sequence

- A generalized ADT that includes all methods from vector and list ADTs
  - plus the following two "bridging" methods that provide connections between ranks and positions:
  - atRank(r): Return the position of the element with rank r.
  - rankOf(p): Return the rank of the element at position p.
- Provides access to its elements from both rank and position
- Can be implemented with an array or doubly linked list

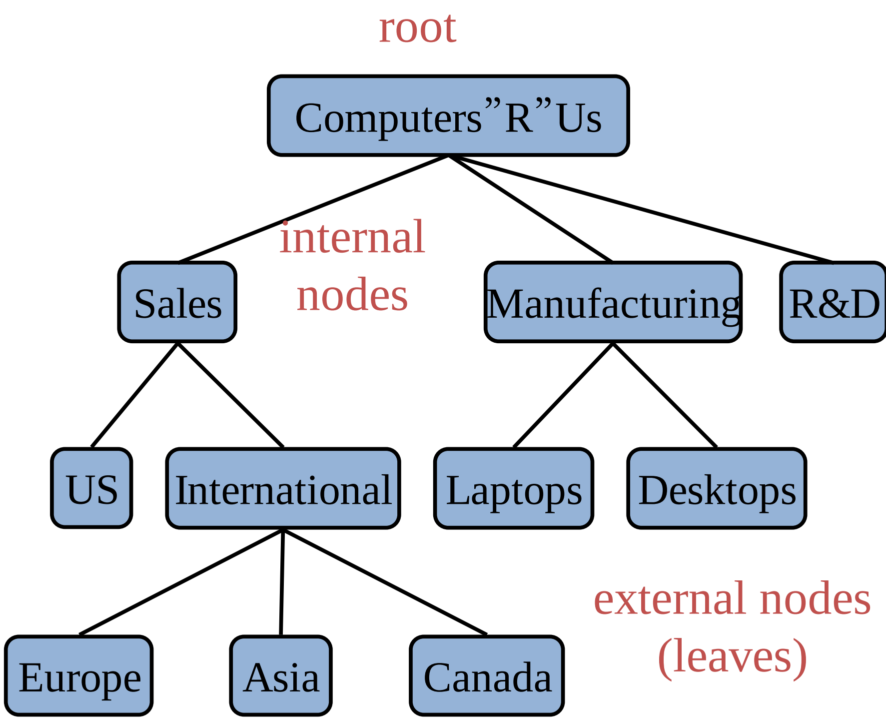
Operation	**Array**	**List**
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$\boldsymbol{O(1)}$	$O(n)$
first, last, before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$\boldsymbol{O(1)}$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast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

Tree

- Stores elements hierarchically

- A tree T is a set of nodes storing elements in a parent-child relationship with the following properties:
  - T has a special node r , called the root of T .
  - Each node v of T different from r has a parent node u.

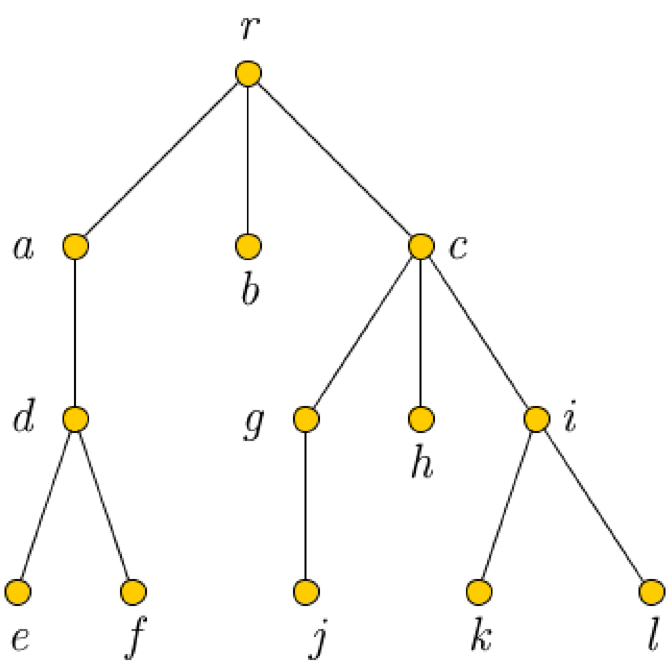
- Direct applications:
  - Organizational charts
  - File systems
  - Programming environments



Tree

- If node u is the parent of node v, then we say that v is a child of u .
- Two nodes that are children of the same parent are siblings .
- A node is external (leaf) if it has no children, and it is internal if it has one or more children.
- The **ancestors** of a vertex are the vertices in the path from the root to this vertex.
- The **descendants** of a vertex *v* are those vertices that have *v* as an ancestor.
- **Depth** : The depth of a node is the number of edges from the node to the tree's root node. In other words, the depth of v is the number of ancestors of v.
- The **height** of a tree T is equal to the maximum depth of an external node of T.
- **Height of a node $\boldsymbol{v}$** is the number of edges on the *longest path* from v to a leaf. A leaf node will have a height of 0. The height of a tree is the largest level of the vertices of a tree which is he height of a root.

Example



- The parent of ***d*** is ***a***.
- The children of ***c*** are ***g, h,*** and ***i*** .
- The siblings of ***g*** are ***h*** and ***i***.
- The ancestors of ***f*** are ***d, a,*** and ***r***.
- The descendants of ***a*** are ***d, e***, and ***f***.
- The internal vertices are ***r, a, d, c, g,*** and ***i*** .
- The leaves are ***e, f, b, j, h, k,*** and ***l***.
- The height of ***d*** is 1.
- The height of ***c*** is 2.
- The height of ***b*** is 0.
- The height of ***r*** is 3 which is the height of tree.
- The depth of ***d*** is 2.
- The depth of ***r*** is 0.
- The depth of ***k*** is 3.
- The height of Tree is 3.

Tree ADT

- Accessor methods :
  - root( ) : Return the root of the tree. O(1)
  - parent(v) : Return the parent of node v; an error occurs if v is root. O(1)
  - children(v) : Return the children of node v. O(c_v)

- Query methods (All takes O(1)):
  - isInternal(v) : Test whether node v is internal.
  - isExternal(v) : Test whether node v is external.
  - isRoot(v) : Test whether node v is the root.

Tree ADT

- Generic methods:
  - size( ) : Return the number of nodes in the tree. O(1)
  - elements( ) : Return an iterator of all the elements stored at nodes of the tree. O(n)
  - positions( ) : Return an iterator of all the nodes of the tree. O(n)
  - swapElements(v, w): Swap the elements stored at the nodes v and w. O(1)
  - replaceElement (v, e) : Replace with e and return the element stored at node v. O(1)

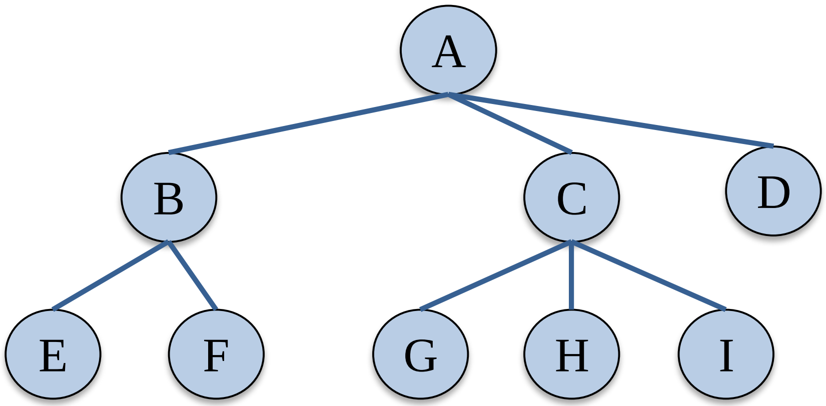
Depth of Tree

- Find the depth of a node v:

```
Algorithm depth(T, v):
    if T. isRoot(v) then
        return 0
    else
        return 1 + depth (T, T. parent(v))
```

- The running time of algorithm depth(T, v) is $O(1 + d_v)$, where d_v denotes the depth of the node v in the tree T.
- Run time is O(n) in the worst-case.

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

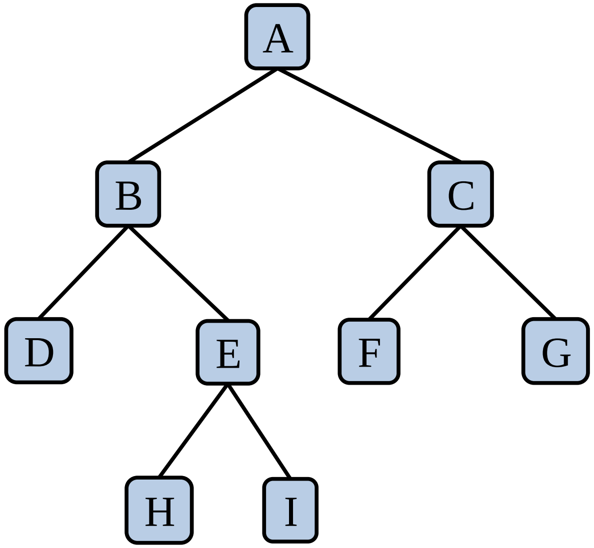
- postorder: a node is visited after its descendants

$O(n)$

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

postOrder(A) visits EFBGHICDA

Binary Trees

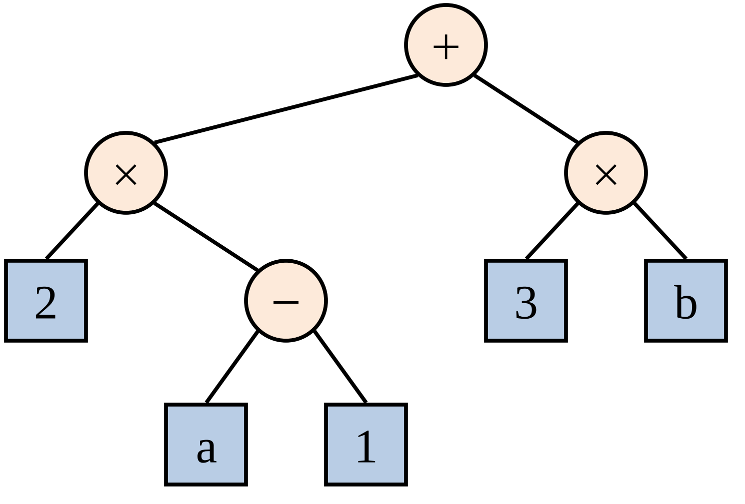


- A binary tree is an ordered tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair (left child, right child)
- Recursive definition: a binary tree is
  - A single node is a binary tree
  - Two binary trees connected by a root is a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

Arithmetic Expression Tree

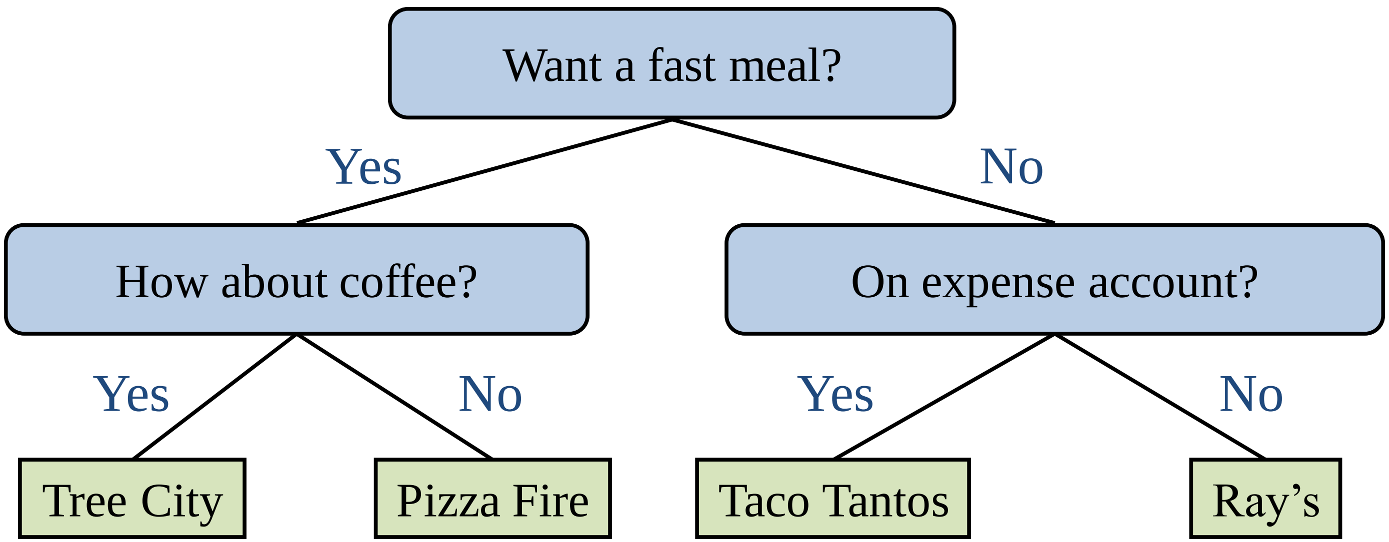
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands

- 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

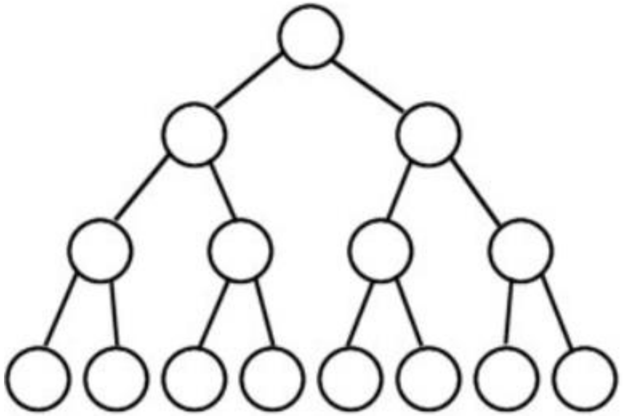


Binary Tree ADT

- Additional accessor methods:
  - leftChild(v): Return the left child of v; an error condition occurs if v is an external node.
  - rightChild(v): Return the right child of v; an error condition occurs if v is an external node.
  - sibling(v): Return the sibling of node v; an error condition occurs if v is the root.

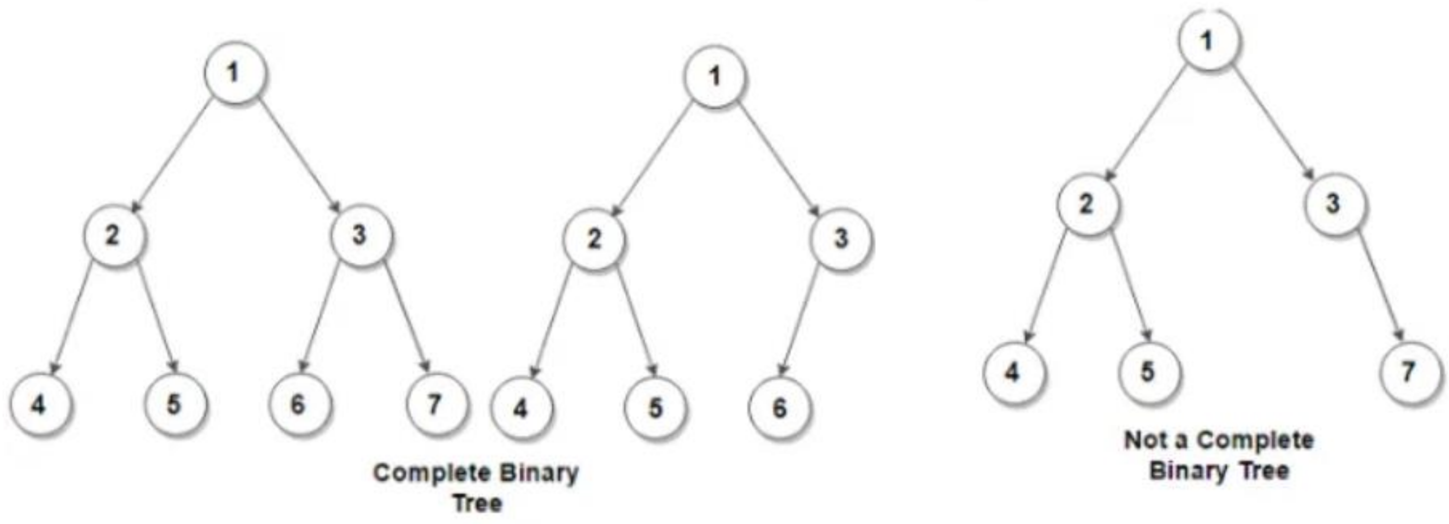
Full Binary Tree

A full binary tree is a tree in which every node other than the leaves has two children.



Complete Binary Tree

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Number of nodes at Levels

- Level l has at most 2^l nodes
- The number of external nodes in T is at least h + 1 and at most 2^h .

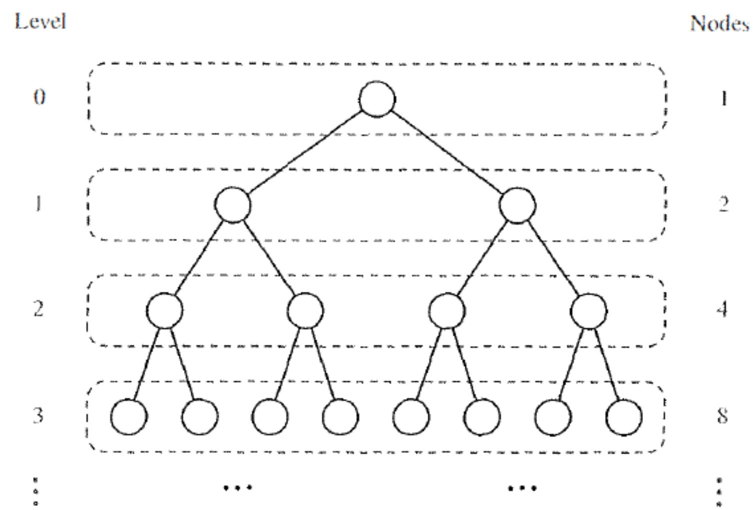


Figure 2.25: Maximum number of nodes in the levels of a binary tree.

Mathematical Review

- Geometric Summation:

$$1 + 2 + 4 + 8 + 16 + \cdots + 2^{n-1} = 2^n - 1$$

$$2^0 + 2^1 + + 2^2 + \cdots 2^{n-1} = 2^n - 1$$

- Another important Summation:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + (n-1) + n$$

$$= \frac{n(n+1)}{2}$$

Total Number of Nodes in Tree

- The total number of nodes in T is :

$$2^0 + 2^1 + +2^2 + \cdots 2^h = 2^{h+1} - 1.$$

Height of Tree

- The height of tree is:

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

So,

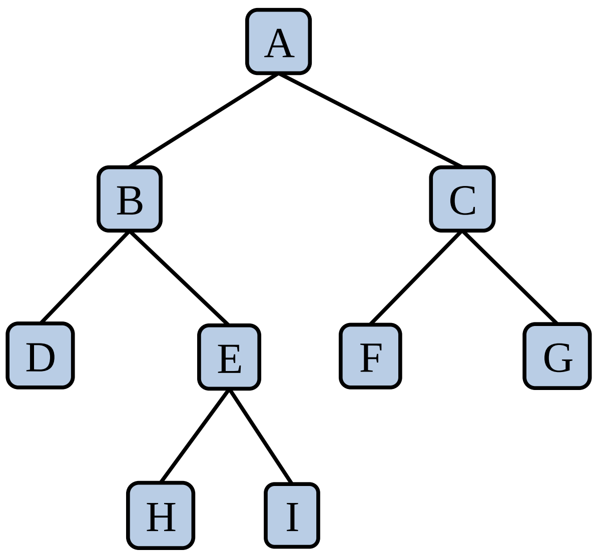
$$h = \log_2(n+1) - 1$$

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*))**



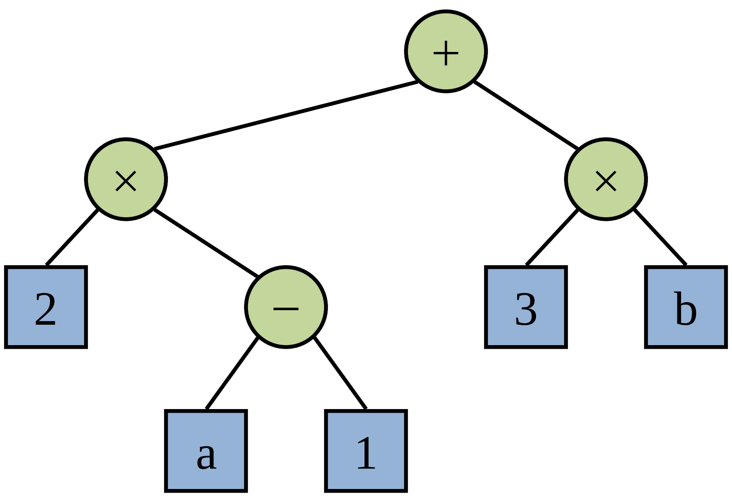
Ex: DBHEIAFCG

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* (")")**

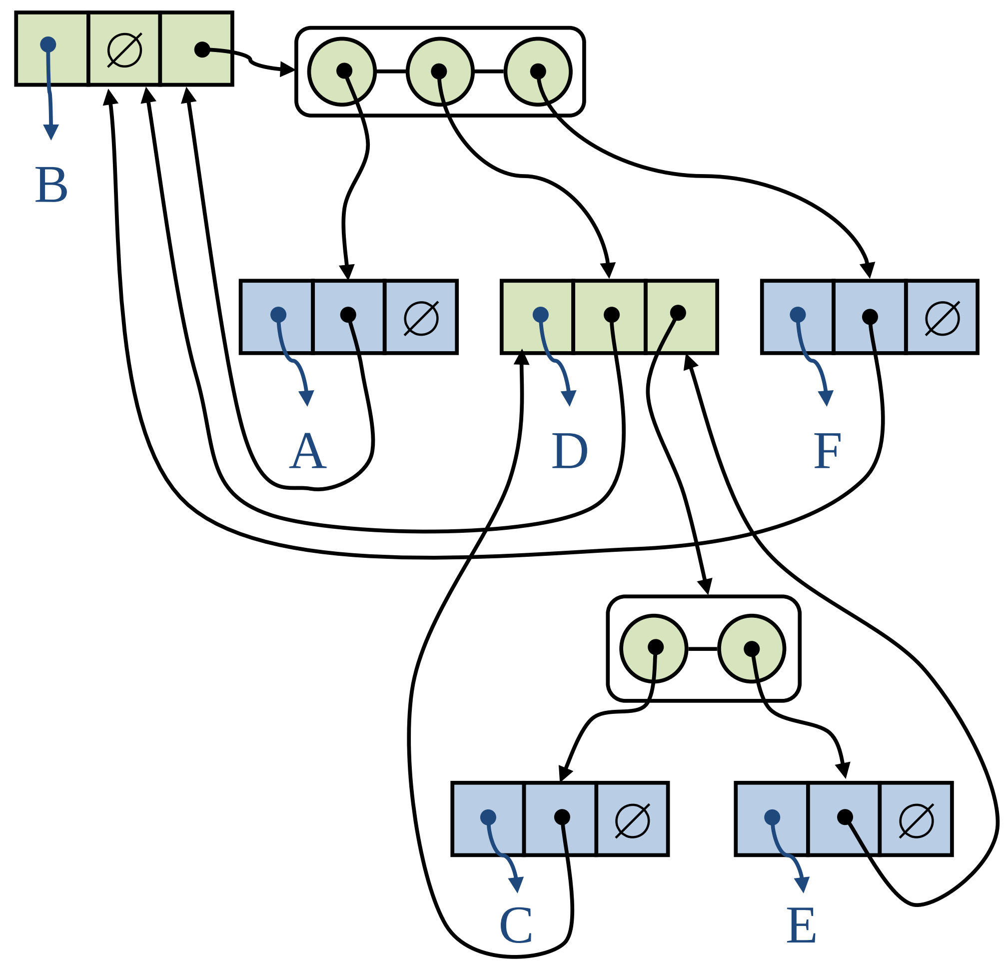


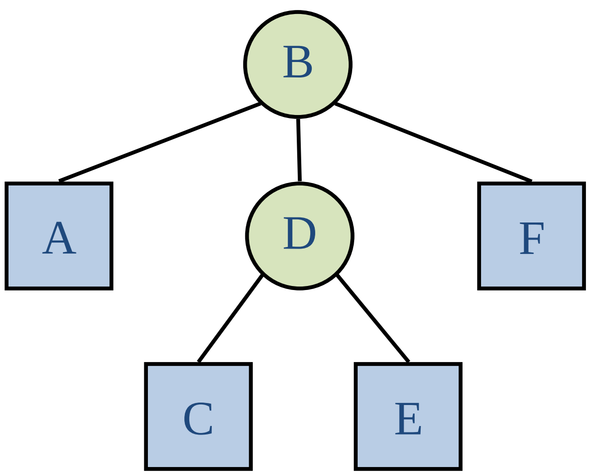
$((2 \times (a - 1)) + (3 \times b))$

Linked Data Structure for Representing Trees

A node stores:

- element
- parent node
- sequence of children nodes

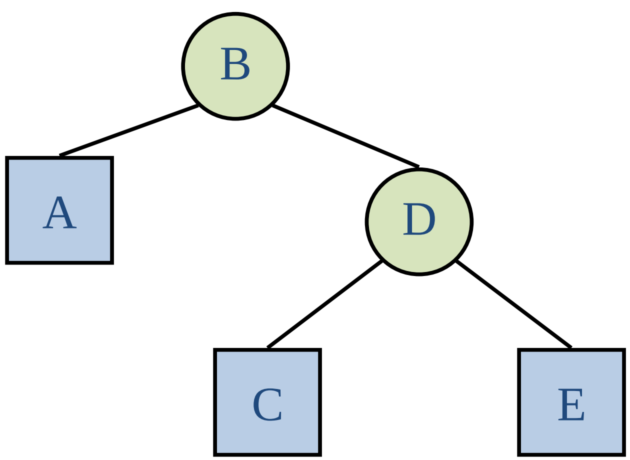


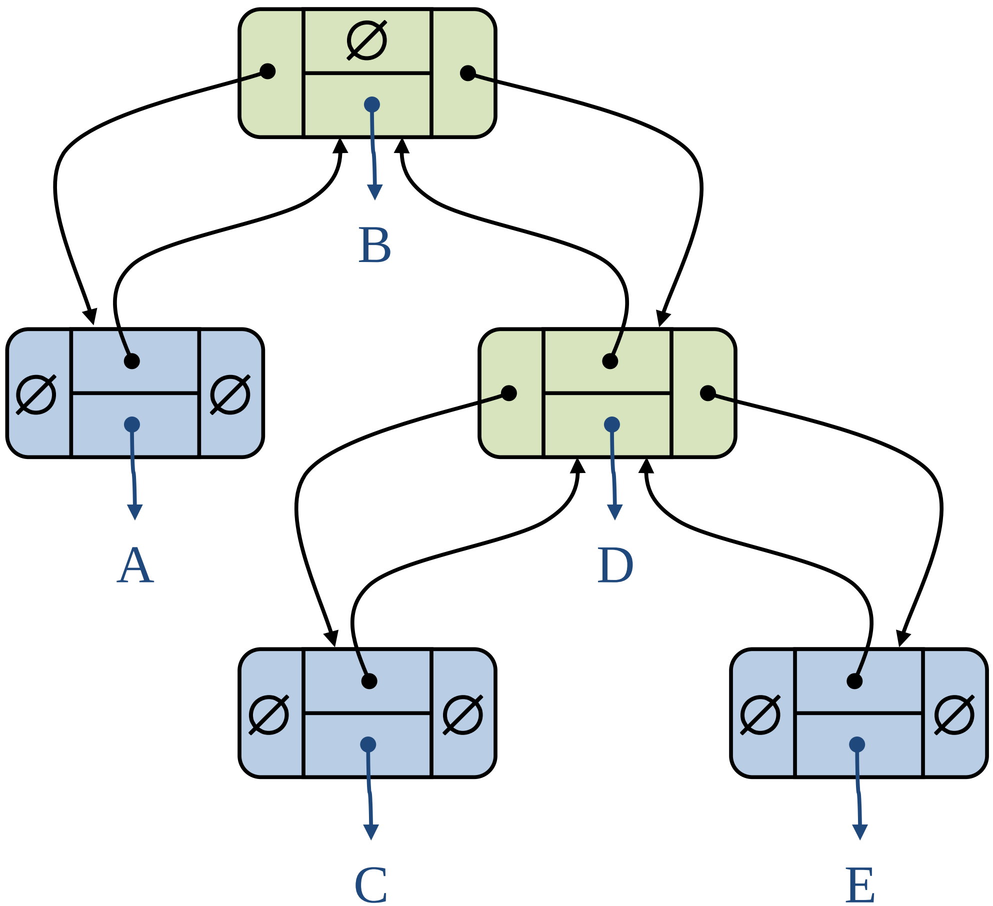


Linked Data Structure for Binary Trees

A node stores:

- element
- parent node
- left node
- right node





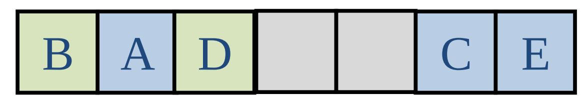
Array-Based Representation of Binary Trees

Nodes are stored in an array

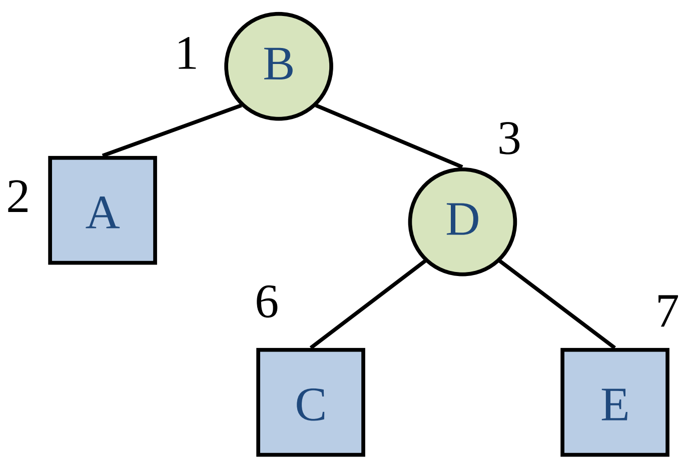
- *rank*(root) = 1
- If *rank*(node) = *i*, then

  rank(leftChild) = 2**i*

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



rank: 1 2 3 4 5 6 7



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

Running Times of BT Operations

Table 2.36 summarizes the running times of the methods of a binary tree implemented with a vector. In this table, we do not include any methods for updating a binary tree.

Operation	**Time**
positions, elements	$O(n)$
swapElements, replaceElement	$O(1)$
root, parent, children	$O(1)$
leftChild, rightChild, sibling	$O(1)$
isInternal, isExternal, isRoot	$O(1)$