Basic Data Structures

CLRS 10.1, 10.2, 10.4
(+ some supplemental material on trees)

Stacks

Queues

Linked Lists

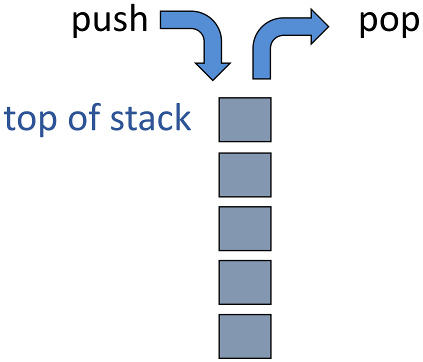
Rooted Trees

Abstract Data Types (ADTs): typical operations

- Search(S, k)
- Insert(S, x)
- Delete(S, x)
- Minimum(S)
- Maximum(S)
- Successor(S,x)
- Predecessor(S,x)

Any specific application will usually require only a few of these to be implemented.

Stack



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

```
STACK-EMPTY(S)
if $S.top == 0$
    return TRUE
else return FALSE
```

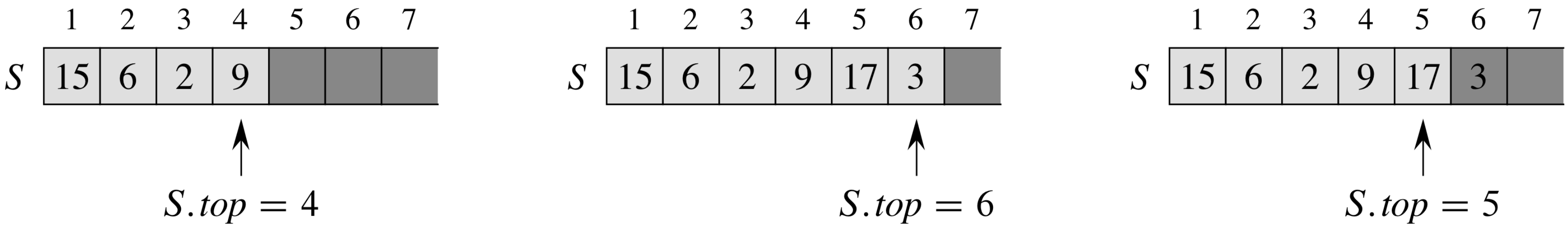
$O(1)$

```
PUSH(S, x)
$S.top = S.top + 1$
$S[S.top] = x$
```

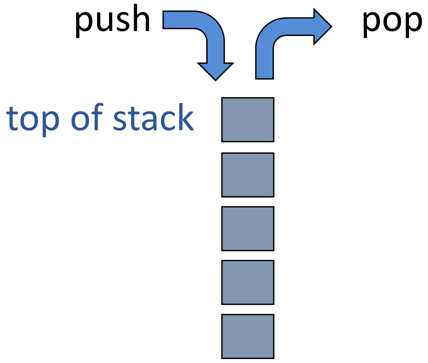
$O(1)$

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else $S.top = S.top - 1$
    return $S[S.top + 1]$
```

$O(1)$



Applications of stack



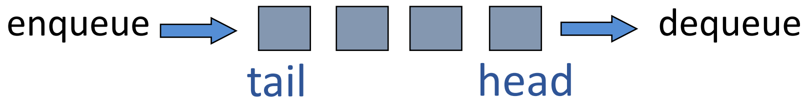
Direct

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

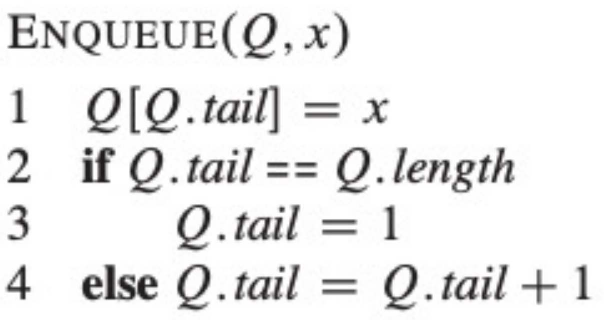
Indirect

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

Queue

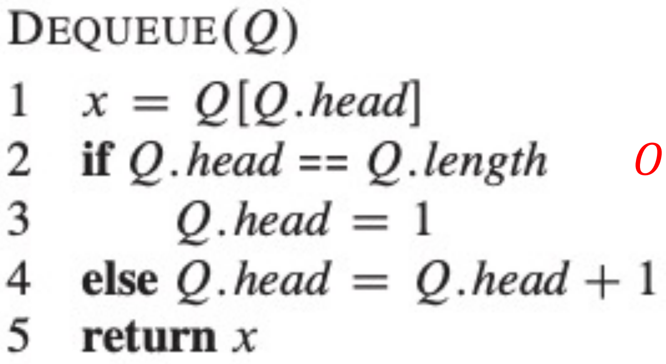


- Container that stores arbitrary objects
- Insertions and deletions follow first-in first-out (FIFO) scheme



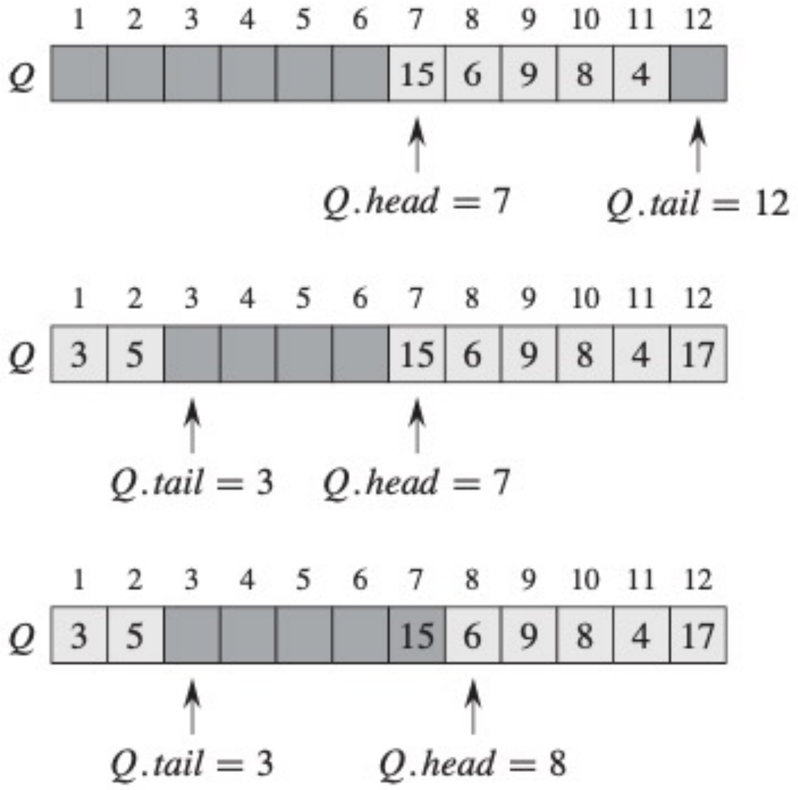
```
ENQUEUE($Q, x$)
    $Q[Q.tail] = x$
    if $Q.tail == Q.length$
        $Q.tail = 1$
    else $Q.tail = Q.tail + 1$
```

$O(1)$

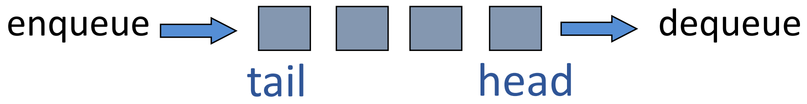


```
DEQUEUE($Q$)
    $x = Q[Q.head]$
    if $Q.head == Q.length$
        $Q.head = 1$
    else $Q.head = Q.head + 1$
    return $x$
```

$O(1)$



Applications of queue



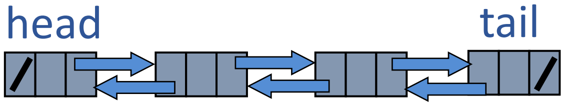
Direct

- Waiting lines
- Access to shared resources
- Multiprogramming

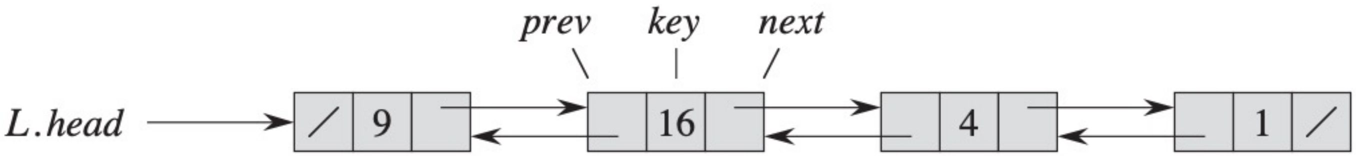
Indirect

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

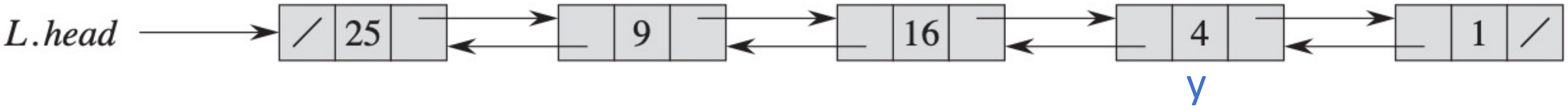
Linked List



- A data structure consisting of a sequence of nodes, each of which stores an element.
- **Singly** linked lists have nodes that contain only a link to the next node
- **Doubly** linked lists have nodes that contain a link to the previous and next node

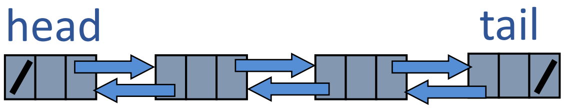


List-Insert(L, x) where x.key = 25



List-Delete(L, y) where y is the object with key 4

Linked List



LIST-SEARCH(L, k) $\Theta(n)$

```
$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return $x$
```

LIST-INSERT(L, x) $\Theta(1)$

```
$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL
```

Note: We are inserting a **node *x*** which has three attributes:

- element value
- previous
- next

LIST-DELETE(L, x) $\Theta(1)$

```
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$
```

Note: Deleting a given **node *x*** requires that we are provided and therefore already have found the node *x*

Deleting an element with a given **key** is $\Theta(n)$ time.

Storing a sequence of items: linked list or array?

- Arrays provide access by **rank** (number of elements preceeding it)
- Linked lists provide access by **position** (node element itself)

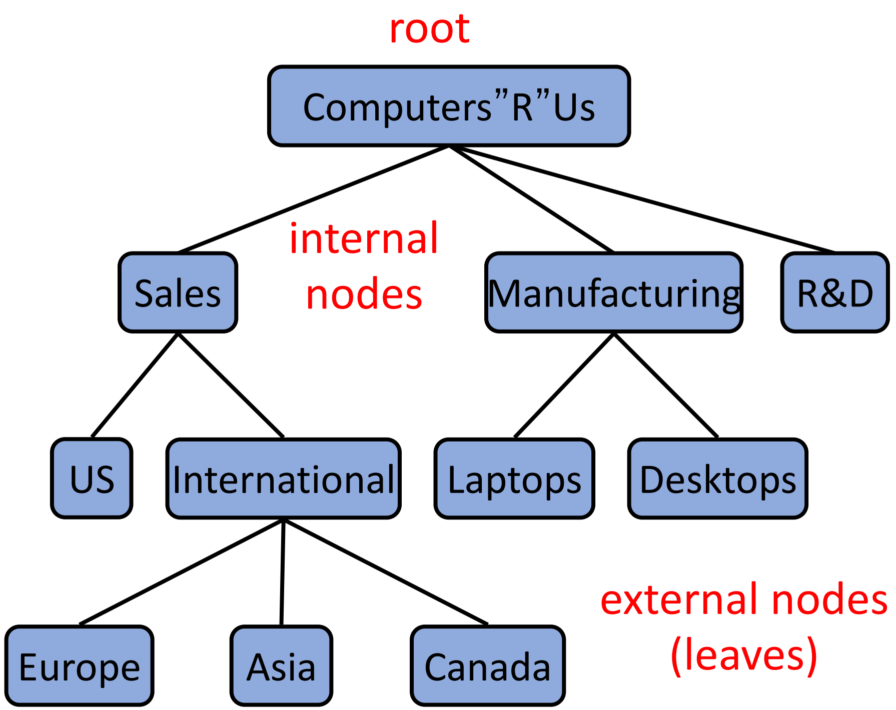
- A sequence can be implemented using either data type
    - Rank-based operations are faster using an array
    - Position-based operations are faster on a linked-list

- Ex: Access element at rank *i*
    - O(1) in array
    - O(n) in a linked list

- Ex: Remove an item at position *p*
    - O(n) in an array
    - O(1) in a linked-list

Rooted trees

- Stores elements hierarchically
- Nodes have a parent-child relationship
- A distinguished node is the **root** of the tree: the only element with no parent
- **External node (leaf)**: a node with no children
- **Internal node**: a node with at least one child



Direct applications

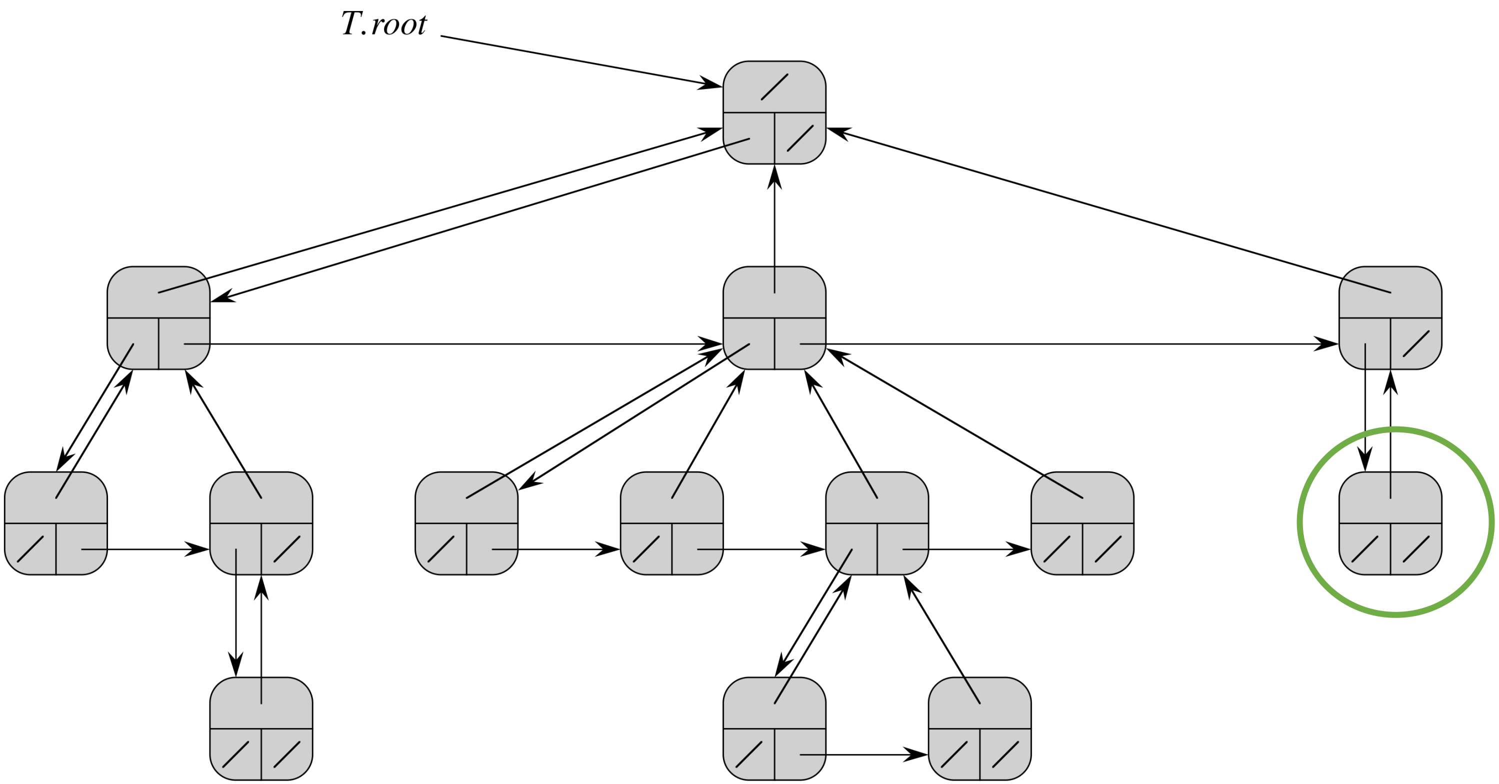
- Organizational charts
- File systems
- Programming environments

Indirect applications

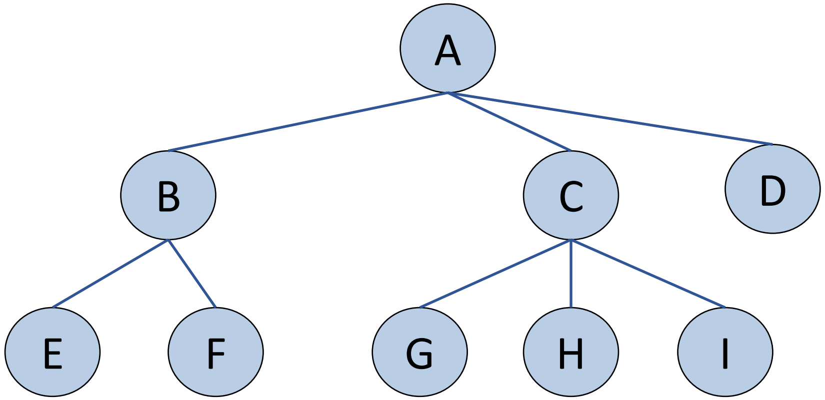
- Component of other data structures

Rooted trees

- Siblings are stored as a linked list (have a pointer to the next sibling)
- **Depth (or level) of a node:** distance to the root
  - Ex: depth of circled node is 2
- **Height**: the maximum depth of any node
  - Ex: height of the tree below is 3



Tree Traversal



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

- In a **preorder** traversal, a node is visited *before* its descendants.

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

preOrder(T, T.root) visits ABEFCGHID

$O(n)$ total time to visit every node

- In a **postorder** traversal, a node is visited *after* its descendants.

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

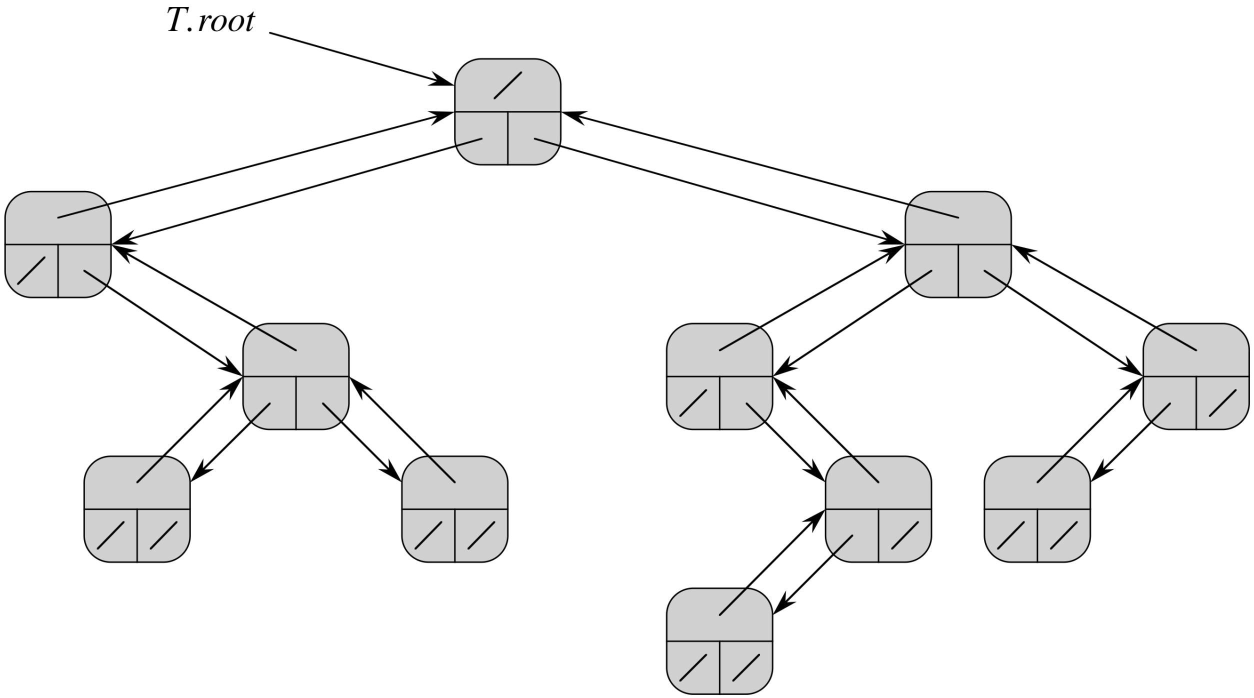
postOrder(T, T.root) visits EFBGHICDA

$O(n)$ total time to visit every node

Binary rooted trees

- **Binary trees**: rooted trees in which each node can have **at most two children**.
    - Children are an ordered pair (left, right)

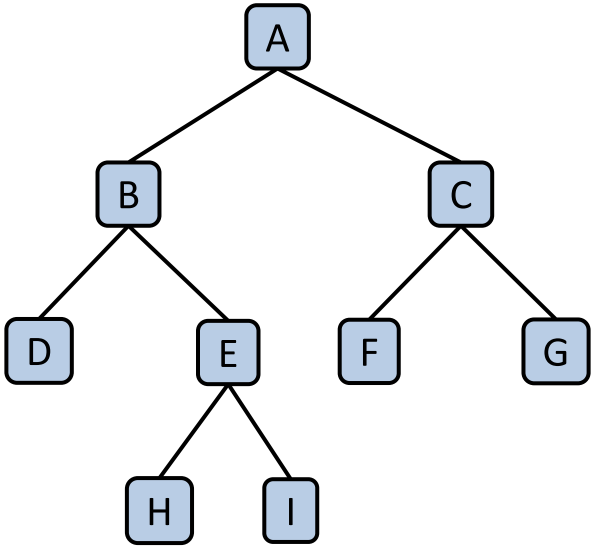
- Applications:
    - arithmetic expressions
    - decision processes
    - searching



Inorder traversal of a binary tree

- In an **inorder** traversal, each node is visited after its left subtree and before its right subtree.

```
Algorithm inOrder(T, v)
    if v has a left child
        inOrder(T, v.left)
    visit(v)
    if v has a right child
        inOrder(T, v.right)
```



inOrder(T, T.root) visits DBHEIAFCG

$O(n)$ total time to visit every node

Euler tour traversal of a binary tree

- 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*) + x 2 – 5 1 x 3 2
  - from below (*inorder*) 2 x 5 – 1 + 3 x 2
  - on the right (*postorder*) 2 5 1 – x 3 2 x +