Basic Data Structures: Arrays and Linked Lists

Array

Contiguous area of memory consisting of equal-size elements indexed by contiguous integers

Property of array

Constant-time access: `array_addr + elem_size*(i - first_index)`

Constant-time to add or remove at the end

Linear time to add or remove at any other location

Times for common operations of array

	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$

Singly-Linked List

Each node contains: `key` and `next` pointer

List API

API	Description	Time Complexity
`PushFront(Key)`	add to front	$O(1)$
`Key TopFront()`	return front item	
`PopFront`	remove front item	$O(1)$
`PushBack(Key)`	add to back	No tail: $O(n)$ With tail: $O(1)$
`Key TopBack()`	return back item	

API	Description	Time Complexity
`PopBack()`	remove back item	No tail: $O(n)$ With tail: $O(n)$ *because it needs a linear search to find the second last node*
`Boolean Find(Key)`	is key in list?	
`Erase(Key)`	remove key from list	
`Boolean Empty()`	empty list?	
`AddBefore(Node, Key)`	adds key before node	
`AddAfter(Nodem Key)`	adds key after node	

```
PushFront(key):
node <- new node
node.key <- key
node.next <- head
head <- node
if tail == nil:
    tail <- head
```

```
PopFront():
if head == nil:
    ERROR: empty list
head <- head.next
if head == nil:
    tail <- nil
```

```
PushBack(key):
node <- new node
node.key <- key
node.next = nil
if tail == nil:
    head <- tail <- node
else:
    tail.next <- node
    tail <- node
```

```
PopBack():
if head == nil:
    ERROR: empty list
if head == tail:
    head <- tail <- nil
else:
    p <- head
    while p.next.next != nil:
        p <- p.next
    p.next <- nil
    tail <- p
```

```
AddAfter(node, key):
node2 <- new node
node2.key <- key
node2.next = node.next
node.next = node2
if tail == node:
    tail <- node2
```

Doubly-Linked List

Each node contains: `key`, `next` pointer and `prev` pointer.

```
PushBack(key):
node <- new node
node.key <- key
node.next = nil
if tail == nil:
    head <- tail <- node
    node.prev <- nil
else:
    tail.next <- node
    node.prev <- tail
    tail <- node
```

```
PopBack(): # O(1) for doubly-linked list and O(n) for singly-linked list
if head == nil:
    ERROR: empty list
if head == tail:
    head <- tail <- nil
else:
    tail <- tail.prev
    tail.next <- nil
```

```
AddAfter(node, key):
node2 <- new node
node2.key <- key
node2.next <- node.next
node2.prev <- node
node.next <- node2
if node2.next != nil:
    node2.next.prev <- node2
if tail == node:
    tail <- node2
```

```
AddBefore(node, key): # O(1) for doubly-linked list and O(n) for singly-
linked list
node2 <- new node
node2.key <- key
node2.next <- node
node2.prev <- node.prev
node.prev <- node2
if node2.prev != nil:
    node2.prev.next <- node2
if head == node:
    head <- node2
```

Remark: With doubly-linked list, constant time to insert between nodes or remove a node. List elements need not be contiguous. It takes $O(n)$ time to find arbitrary element.

Basic Data Structures: Stacks and Queues

Stack

An abstract data type with the following operations:

- `Push(Key)`: adds key to collection
- `Key Top()`: returns most recently-added key
- `Key Pop()`: removes and returns most recently-added key
- `Boolean Empty()`: are they any elements?

Balanced Brackets

Input: A string `str` consisting of '(', ')', '[', ']' characters.

Output: Return whether or not the string's parentheses and square brackets are balanced.

```
IsBalanced(str):
Stack stack
for char in str:
    if char in ['(', '[']:
        stack.Push(char)
    else:
        if stack.Empty():
            return False
        top <- stack.Pop()
        if (top = '[') and char != ']') or (top = '(') and char != ')'):
            return False
return stack.Empty()
```

Stacks can be implemented with either an array or a linked list. Each stack operation is $O(1)$: Push, Pop, Top, Empty. Stacks are occasionally known as LIFO queues.

Queue

An abstract data type with the following operations:

- `Enqueue(Key)`: adds key to collection
- `Key Dequeue()`: removes and returns least recently-added key
- `Boolean Empty()`: are there any elements?

Queues can be implemented with either a linked list (with tail pointer) or an array (Dequeue costs $O(n)$ under array implementation). Each queue operation is $O(1)$: Enqueue, Dequeue, Empty. Queues are FIFO data structures.

Basic Data Structures: Trees

Tree

A tree is

- empty, or
- a node with:
    - a key, and
    - a list of child trees.

Terminology of tree

- Root: top node in the tree
- A child has a line down directly from a parent
- Ancestor: parent, or parent of parent, etc.
- Descendant: child, or child of child, etc.
- Sibling: sharing the same parent
- Leaf: node with no children
- Interior node: non leaf
- Level: 1 + num edges between root and node
- Height: maximum depth of subtree node and farthest leaf

- Forest: collection of trees

In general, node contains:

- key
- children: list of children nodes
- (optional) parent

For binary tree, node contains:

- key
- left
- right
- (optional) parent

```
Height(tree):
if tree == nil:
    return 0
return 1 + Max(Height(tree.left), Height(tree.right))
```

```
Size(tree):
if tree == nil:
    return 0
return 1 + Size(tree.left) + Size(tree.right)
```

```
InOrderTraversal(tree): # Depth first
if tree == nil:
    return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

```
PreOrderTraversal(tree): # Depth first
if tree == nil:
    return
Print(tree.key)
PreOrderTraversal(tree.left)
PreORderTraversal(tree.right)
```

```
PostOrderTraversal(tree): # Depth first
if tree == nil:
    return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```

```
LevelTraversal(tree): # Breadth first
if tree == nil:
    return
Queue q
q.Enqueue(tree)
while not q.Empty():
    node <- q.Dequeue()
    Print(node)
    if node.left != nil:
        q.Enqueue(node.left)
    if node.right != nil:
        q.Enqueue(node.right)
```