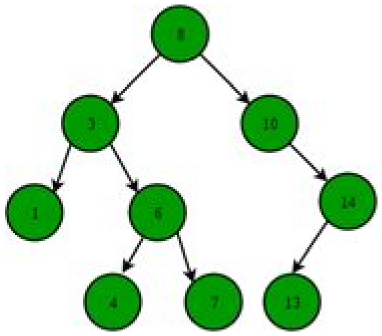
ADT

Binary Search Tree ADT



{Inv: For any node n, each node in the left subtree of n has a key < n's key, and every node in the right subtree of n has a key > n's key; Root != null.}

Primitive Operations:		
BinarySearchTree		→ BinarySearchTree
addNode	Root x K x V	→ BinarySearchTree
updateNode	Root x K x V	→ BinarySearchTree
searchNode	Root x K	→ Node
deleteNode	Root x K	→ BinarySearchTree
inOrder	Root x Collection	→ Collection
preOrder	Root x Collection	→ Collection
postOrder	Root x Collection	→ Collection
getWeight	Root	→ Int
getHeight	Root	→ Int

BinarySearchTree()
“Creates a new binary search tree”
{pre: TRUE}
{post: binarySearchTree = {Root: null, Weight: 0, Height: 0}}
Primitive Operation: Constructor

addNode(root, k, v)
“Adds a new node to the binary search tree”
{pre: k, v ∈ Object}
{post: <binarySearchTree>}
Primitive Operation: Modifier

updateNode(root, k, v)
“Updates the Value v of the node with Key k”
{pre: node = {…, V: v’, …} ∧ k, v ∈ Object}
{post: node.V = v, <binarySearchTree>}
Primitive Operation: Modifier

searchNode (root, k) “Searches a node in the binary search tree” {pre: k ∈ Object} {post: <node>} Primitive Operation: Analyzer
deleteNode (root, k) “Deletes a node from the binary search tree” {pre: k ∈ Object} {post: <binarySearchTree>} Primitive Operation: Modifier
inOrder (root, collection) “Orders the binary search tree inorder” {pre: binarySearchTree.Root != null} {post: <collection> ordered in inorder>} Primitive Operation: Modifier

preOrder(root, collection)
“Orders the binary search tree preorder”
{pre: binarySearchTree.Root != null}
{post: <collection> ordered in preorder>}
Primitive Operation: Modifier

postOrder(root, collection)
“Orders the binary search tree postorder”
{pre: binarySearchTree.Root != null}
{post: <collection> ordered in postorder>}
Primitive Operation: Modifier

getWeight(root)
“Calculates the weight of the binary search tree”
{pre: binarySearchTree.Root != null}
{post: <weight>}
Primitive Operation: Analyzer

getHeight(root)
“Calculates the height of the binary search tree”
{pre: binarySearchTree.Root != null}
{post: <height>}
Primitive Operation: Analyzer

Stack ADT

Stack= $< \ll a1, a2, a3, a4..., an \gg, top >$

{inv: 0≤n ∧ Size(Stack)=n ∧ top=a_n}

Constructor Operations:

● CreateStack		⟶ Stack
● Push	Stack x Element	⟶ Stack
● Peek	Stack	⟶ Element
● Pop	Stack	⟶ Element
● IsEmpty	Stack	⟶ Boolean
● Size	Stack	⟶ Integer

CreateStack()

“Builds an empty stack”

{pre: TRUE}

{pos: stack $\neq$ Ø}

Primitive Operation: Constructor

Push(E element)

“Adds a new element ‘e’ to the Stack”

{pre: stack = $\ll a1, a2, a3, a4..., an \gg$ v stack = Ø}

{pos: stack $\neq$ Ø}

Primitive Operation: Modifier

Peek()
“Shows the top of the stack”
{pre: stack = $\ll a1, a2, a3, a4..., \ an \gg$ v stack = Ø}
{pos: a_n v NoSuchElementException}
Primitive Operation: Analyzer

Pop()
“Shows the top of the stack and deletes it”
{pre: stack = Ø v stack = $\ll a1, a2, a3, a4..., \ an \gg$ }
{pos: NoSuchElementException v (a_n ∧ stack = $\ll a1, a2, a3, a4..., \ an-1 \gg$)}
Primitive Operation: Modifier

IsEmpty()
“Determines if a stack is empty or not”
{pre: stack}
{pos: true if stack = Ø v false if stack $\neq$ Ø}
Primitive Operation: Analyzer

Size()
“Shows the current size of the stack”
{pre: stack $\neq$ Ø }
{pos: size = Size(stack)}
Primitive Operation: Analyzer

Queue ADT

Queue = $< \ll a1, a2, a3, a4..., \ an \gg, \ head, \ tail >$

{inv: 0≤n ∧ Size(Queue)=n ∧ head=a_1 ∧ tail=a_n}

Constructor Operations:

CreateQueue		⟶ Queue
Enqueue	Queue x Element	⟶ Queue
Peek	Queue	⟶ Element
Dequeue	Queue	⟶ Element
IsEmpty	Queue	⟶ Boolean
Size	Queue	⟶ Integer

CreateQueue()
“Builds an empty queue”
{pre: TRUE}
{pos: queue $\neq$ Ø}
Primitive Operation: Constructor

Enqueue(E element)
“Adds a new element ‘e’ to the Queue”
{pre: queue = $\ll a1, a2, a3, a4..., \ an \gg$ v queue = Ø }
{pos: queue $\neq$ Ø}
Primitive Operation: Constructor

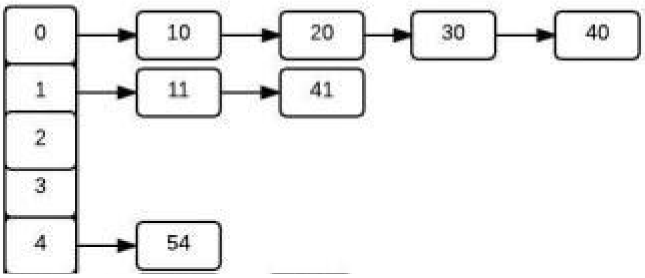
Peek()
“Retrieves the head of the queue”
{pre: queue = $\ll a1, a2, a3, a4..., \ an \gg$ v queue = Ø }
{pos: a_1 v NoSuchElementException}
Primitive Operation: Analyzer

Dequeue()
“Retrieves and deletes the head of the queue”
{pre: queue= Ø v stack= $\ll a1, a2, a3, a4..., an \gg$ }
{pos: NoSuchElementException v (a_1 $\wedge$ queue= $\ll a2, a3, a4..., an \gg$)}
Primitive Operation: Modifier

IsEmpty()
“Determines whether a queue is empty or not”
{pre: queue}
{pos: true if queue = Ø v false if queue $\neq$ Ø}
Primitive Operation: Analyzer

Size()
“Returns the current size of the queue”
{pre: queue $\neq$ Ø}
{pos: size= Size(stack)}
Primitive Operation: Analyzer

HashTable ADT



{inv: hash(k)=i}

Constructor Operations:

● createTable		⟶ HashTable
● Insert	HashTable x Item	⟶ HashTable
● hashFunction	HashTable x Key	⟶ Integer
● Delete	Key	⟶ HashTable
● get	Key	⟶ Item
● getSize()	HashTable	⟶ Integer

CreateHashTable()

“Builds an empty hash”

{pre: TRUE}

{pos: hash $\neq$ Ø}

Primitive Operation: Constructor

Insert(K k, V v)

“Adds a new element ‘v’ to the hash, with its proper position key”

{pre: hash= $\ll a1, a2, a3, a4..., an \gg$ v hash= Ø }

{pos: hash $\cup$v ^ v is sorted}

Primitive Operation: Modifier

HashFunction(K k)

“Gets the function that sorts the hash to get the key’s element at its proper place, evading collisions”

{pre: k = Ø }

{pos: index $=$ proper position to the key}

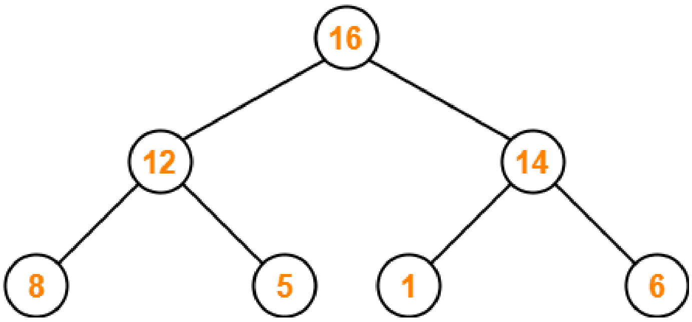
Primitive Operation: Modifier

Delete(K k)
“Deletes an element of the hash”
{pre: hash= $\ll a1, a2, a3, a4..., \ an \gg$ v hash= Ø }
{pos: k $\notin hash$ }
Primitive Operation: Modifier

getSize()
“Gets the size of the hash”
{pre: hash= $\ll a1, a2, a3, a4..., \ an \gg$ }
{pos: size= hash.length}
Primitive Operation: Analyzer

get(K k)
“Gets the element by searching it with its key”
{pre: hash= $\ll a1, a2, a3, a4..., \ an \gg$ v hash= Ø }
{pos: hash[k]= V v hash[k]= null}
Primitive Operation: Analyzer

Heap ADT



{inv: A[parent(index)] >= A[i]}

Constructor Operations:

createHeap		→ Heap
parent	Index	→ Integer
rightChild	Index	→ Integer
leftChild	Index	→ Integer
heapify	Heap x Index	→ Heap
increaseMaxHeap	Element x Index	→ Heap
extract	Element	→ Heap
max	Element	→ Heap
buildHeap		→ Heap
insert	Heap x Element	→ Heap
isEmpty	Heap	→ Boolean

CreateHeap()

“Builds an empty heap”

{pre: TRUE}

{pos: heap ≠ Ø}

Primitive Operation: Constructor

parent(int index)

“Search parent’s index”

{pre: heap ≠ Ø}

{pos: parent[index]}

Primitive Operation: Constructor

rightChild(int index)

“Search right child of index”
{pre: heap ≠ Ø}
{pos: right[index]}
Primitive Operation: Constructor

leftChild(int index)
“Search left child of index”
{pre: TRUE}
{pos: left[index]}
Primitive Operation: Constructor

heapify(int index)
“Maintains the max-heap property so the heap can remain as max-heap”
{pre: TRUE}
{pos: heap with max-heap property well.}
Primitive Operation: Constructor

buildHeap()
“Produces a max-heap from an unordered array”
{pre: A[1… n] }
{pos: A.length= max-heap }
Primitive Operation: Constructor

increaseMaxHeap(int index, P element)
“Increases the value of element at the index to the new element, adding the element to its proper index to maintain the max-heap property ”
{pre: heap ≠ Ø}
{pos: heap which index has the proper element}

Primitive Operation: Modifier

extract()
“Extracts the max (parent) element of the heap, and applies max-heap propierty to keep it as a max-heap”
{pre: heap ≠ Ø ^ heap= A[n1,n2,n3,n4,n5}}
{pos: heap=<[n2,n3,n4,n5], n1>}
Primitive Operation: Modifier

max()
“Shows the max element in the heap, that it’s located in the first position in the array.”
{pre: heap ≠ Ø}
{pos: heap[1]}
Primitive Operation: Analyzer

insert(P element)
“Inserts the element P in the array of the heap”
{pre: (heap = Ø v heap ≠ Ø) ^ element ≠ null}
{pos: heap ∪ {element} }
Primitive Operation: Modifier

IsEmpty()
“Determines whether a queue is empty or not”
{pre: heap}
{pos: true if heap= Ø v false if heap ≠ Ø}
Primitive Operation: Analyzer