Purpose of Data Structures

Organize data so that it can be accessed quickly and usefully.

Examples: Lists, stacks, queues, heaps, search trees, hash tables, bloom filters, union-find, etc.

Why so many? Different data structures support different sets of operations and are suitable for different types of tasks. For example, when solving a problem using recursion a stack data structure is optimal given the last-in, first-out (LIFO) nature of recursive solutions.

Rule of Thumb: Choose the "minimal" data structure that supports all the operations that you need and no more.

Heap: Supported Operations

A heap is a container for objects which have keys that can be compared. That is, employee records (employee numbers), network edges (weight, length, number of connections, etc.), events (time for which an event happens or in relation to another event), etc.

Insert: Add a new object to a heap.
Running time = $O(\lg n)$, where n = # of objects in heap.

Extract-Min / Extract-Max: Remove an object in the heap with a minimum/maximum key value with ties broken arbitrarily.
Note: Extract-Min or Extract-Max exclusively. A heap supports one or the other, not both. If given Extract-Min, but you need Extract-Max, then negate the values in the heap and proceed as normal. This works similarly if given Extract-Max, but you need Extract-Min.
Running Time = $O(\lg n)$

Heapify: Inserts n batched objects in $O(n)$ time rather than $O(n \lg n)$ time

Delete: Deletes an objects from anywhere in the heap in $O(\lg n)$ time.

(con't on next sheet)

Sorting

Use a heap in sorting based applications as a fast way to do repeated minimum or maximum computations.

Using a heap to improve selection sort

Selection sort is a sorting algorithm which runs in $O(n^2)$ time for an array of length n.

Selection sort works by scanning all elements in an array and placing the smallest element found in the $\emptyset^{th}$ index position, then repeating the scan to find the next smallest element and putting it in the next index position until the entire array is sorted.

Given: $\{8, 4, 1, 5, 2\}$ Find: The sorted array

1. Scan the array for the smallest element, $s = 1$
2. Swap the smallest element into the first position and advance the index marker.

$$\{8, 4, 1, 5, 2\} \xrightarrow{} \{1, 4, 8, 5, 2\}$$

SWAP; $i=\emptyset$ → $i=1$

3. Scan the array from the index marker for the smallest element, $s = 2$
4. Swap the smallest element into the index marker's position and advance the index marker.

$$\{1, 4, 8, 5, 2\} \longrightarrow \{1, 2, 8, 5, 4\}$$

SWAP; $i=1$ → $i=2$

5. Repeat until the array is exhausted. $\{1, 2, 4, 5, 8\}$ $i=4$

How a Heap Data Structure Improves Selection Sort

Heap Sort

1. Insert all n array elements into a heap. Heapify $= O(n)$
2. Continually extract the minimum element until heap is exhausted. Extract-Min $= O(\lg n)$, n times

Running time of HeapSort requires 2 Heap operations

$$O(n) = \underbrace{n \lg n}_{\text{Extract-Min}} + \underbrace{n}_{\text{Heapify}}$$

$\hookrightarrow O(n \lg n)$ optimal for a comparison-based sorting algorithm.

APPLICATION EXAMPLES

SIMULATION

Use of a "priority queue", aka, heap, allows proper scheduling of events that need to occur.

Let objects represent the event records (action/update to occur at a given time in the future.)

Let the keys be the time the event is scheduled to occur

Using EXTRACT-MIN operation continually yields the next scheduled event in the correct order.

MEDIAN MAINTENANCE

GIVEN: A sequence; $x_1, \ldots, x_n$; of numbers, one-by, one.

FIND: The median of $\{x_1, \ldots, x_i\}$ as each i^{th} element is presented in $O(\lg i)$ runtime.

SOL'N:
Can easily be done in $O(i)$ runtime, but violates $O(\lg i)$ constraint.

This can be solved with two heaps:
1) H_{LOW} using EXTRACT-MAX and keeps the lowest half of the i elements

2) H_{HIGH} using EXTRACT-MIN and keeps the highest half of the i elements

Must maintain $i/2$ smallest elements in H_{LOW} and $i/2$ largest elements in H_{HIGH}. This is done by running EXTRACT-MAX or EXTRACT-MIN for the unbalanced heap and putting the extracted element in the other heap to maintain the $i/2$ requirement.

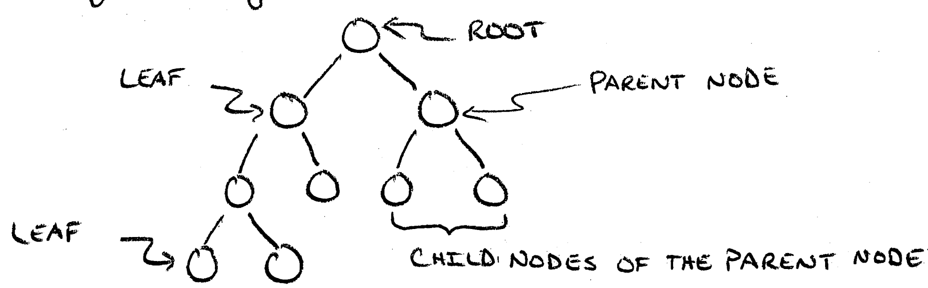
Implementing the above will easily allow the median to be calculated in $O(\lg i)$ runtime as each element is inserted. It's either the minimum of H_{HIGH} the maximum of H_{LOW} or the average of the two depending on the number of elements.

Implementation of Insert and Extract-Min

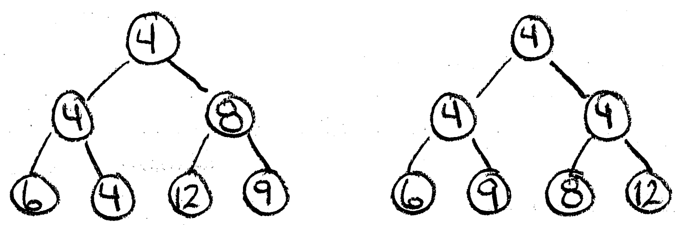
Ways to Visualize a Heap

Binary Tree

Each level of the tree is filled in as completely as possible, with partially filled in levels filled in from left-to-right.



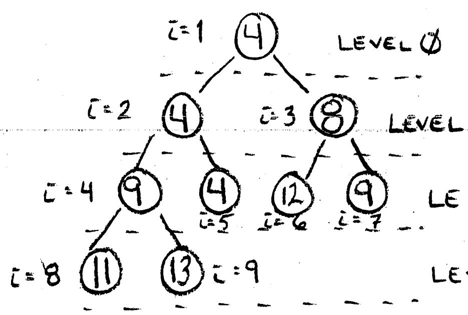
Heap Property: At every node x, Key[x] $\leq$ all keys of x's children



Both are valid representations of the same heap.

As a consequence of arranging a heap in this manner, the minimum value of the heap is always at the root.

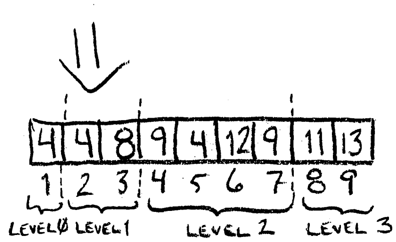
As an Array (1-indexed)



$$\text{Parent}(i) = \begin{cases} i/2 & \text{if } i \% 2 = \emptyset \text{ (even)} \\ \lfloor i/2 \rfloor & \text{if } i \% 2 \neq \emptyset \text{ (odd)} \end{cases}$$

↑ floor, round down

$\text{Children}(i) = 2i$ and $2i + 1$



(con't on next sheet)

IMPLEMENTATION OF INSERT AND EXTRACT-MIN (con't)

INSERT AND BUBBLE-UP

GIVEN: A new key to add to the heap, k.

FIND: Where in the heap k belongs.

SOL'N:

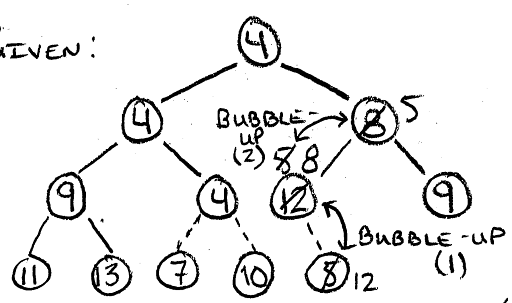
STEP 1: Place k at the end of the last level or create a new level if the last level is full in the tree representation.
Alternatively in the array representation k is appended in the last index position

STEP 2: Bubble-up k until heap property is restored, that is, the key of k's parent is $\leq$ k.
In the array values are swapped appropriately between the parent-child positions!

STEP 3: Continue step 2 until the heap property is restored.

Runtime is at most $O(\lg n)$ since there are $\lg n$ levels to a balanced binary tree and in the worst case k appended to the end may have to bubble all the way up to the root.

GIVEN:



Add 7, Parent(7) = 4, 4 < 7 OK

Add 10, Parent(10) = 4, 4 < 10 OK

Add 5, Parent(5) = 12, 12 > 5 BUBBLE-UP

(1) After BUBBLE-UP
Parent(5) = 8, 8 > 5 BUBBLE-UP

(2) After BUBBLE-UP
Parent(5) = 4, 4 < 5 OK

(con't on next sheet)

IMPLEMENTATION OF INSERT AND EXTRACT-MIN (con't)

EXTRACT-MIN AND BUBBLE-DOWN

GIVEN: A requirement to provide the minimum value in the heap.

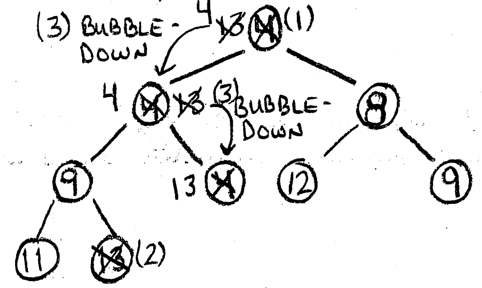
FIND: The minimum value.

SOL'N:
STEP 1: Return the value of the root and delete it.

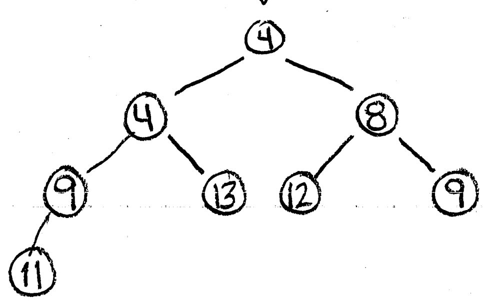
STEP 2: Move the last leaf into the root position.

STEP 3: Iteratively swap the value moved into the root position with the smaller child node (BUBBLE-DOWN) until the heap property has been restored.

Runtime is at most $O(\lg n)$ for the same reason as the INSERT operation on the previous page.



1) Return the value, 4 and remove the root
2) Move the last leaf, 13, into the root position.
3) Swap with smaller of child nodes until heap property (page 4) is restored.



Properly restored heap after EXTRACT-MIN operation.

```python
# heap_sort.py
"""
Example of heapsort implementation using min-heap.

Demonstrates insertion into a heap data structure and extract minimum
value from the heap.
"""

def main():
    A = [11, 13, 9, 4, 12, 9, 4, 8, 4]
    print("Original list: {}".format(A))

    # Create the heap
    heap = heapify(A)

    print("List after heapifying: {}".format(heap))

    sorted_list = []
    # Sort the original list
    while len(heap) > 1:
        min_value, heap = extract_min(heap)
        sorted_list.append(min_value)

    print("The sorted list using heapsort is:")
    print(sorted_list)

def heapify(A):
    """ Turns the input array into an array resembling a heap."""

    heap = []
    # Heap array is a 1-based index, place a filler at the 0 index.
    heap.append(0)

    for key in A:
        insert(key, heap)

    return heap

def insert(key, heap):
    """Inserts a key into the heap."""

    # The first element, whatever it is, gets put in the parent node.
    if len(heap) < 2:
        heap.append(key)
        return

```

```
    heap.append(key)

    bubble_up(len(heap)-1, heap)

    return

def get_parent_index(child_index):
    """
    Get the index of the given key's parent given the index
    of the child key.
    """

    # The root of the tree is at index position 1
    # and can have no parent
    if child_index == 1:
        return 0
    return child_index // 2

def get_left_child_index(parent_index, heap):
    """
    Get the index of the left child given the
    parent node's index.
    """

    # Remember, this is a 1-based index.
    if parent_index * 2 >= len(heap):
        # There is no left child
        return 0

    return parent_index * 2

def get_right_child_index(parent_index, heap):
    """
    Get the index of the right child given the
    parent node's index.
    """

    # Remember, this is a 1-based index.
    if parent_index * 2 + 1 >= len(heap):
        # There is no right child
        return 0

    return parent_index * 2 + 1

def bubble_up(child_index, heap):
    """
```

```
    Push a child node up through the heap to maintain heap property.
    """

    parent_index = get_parent_index(child_index)
    # The node in question is the root node
    if parent_index == 0:
        return

    if heap[child_index] < heap[parent_index]:
        heap[child_index],  heap[parent_index]  =  heap[parent_index],
heap[child_index]
        bubble_up(parent_index, heap)

    return

def extract_min(heap):
    """Returns the minimum value of the heap, the root."""

    # Check if the heap only has the 0-element
    if len(heap) < 2:
        return heap[0], heap

    root = heap[1]
    # Swap the last index position into the root node
    heap[1] = heap[-1]
    # Pare down the list since the last node became the root
    del heap[-1]

    if len(heap) > 1:
        bubble_down(1, heap)

    return root, heap

def bubble_down(parent_index, heap):
    """Moves a misplaced node into its appropriate position."""

    min_val = heap[parent_index]
    min_index = get_left_child_index(parent_index, heap)

    # The parent node has no children
    if min_index == 0:
        return

    # Find the smaller of the two children
    right_child_index = get_right_child_index(parent_index, heap)
    if right_child_index == 0:
```

```
        right_child_index = min_index

    if heap[right_child_index] < heap[min_index]:
        min_index = right_child_index

    if heap[min_index] < min_val:
        min_val = heap[min_index]

    if min_val != heap[parent_index]:
        heap[parent_index], heap[min_index] = heap[min_index], heap[parent_index]
        bubble_down(min_index, heap)

if __name__ == "__main__":
    main()
```

```go
package main

/*
Example of heapsort implementation using min-heap.

Demonstrates insertion into a heap data structure and extraction of the
minimum value from the heap to make a sorted slice.
*/

import "fmt"

type heap struct {
	xi []int
}

func main() {
	fmt.Println("This program provides an example of a min-heap sort.")
	a := []int{11, 13, 9, 4, 12, 9, 4, 8, 4}

	// Create the heap
	h := heap{a}
	fmt.Println("The original list:", h.xi)
	h.Heapify()

	fmt.Println("The slice after heapifying:", h.xi)

	// Sort the heap
	heapLength := h.Len()
	for i := 0; i < heapLength; i++ {
		min, hasNode := h.ExtractMin()
		if hasNode {
			a[i] = min
		}
	}
	fmt.Println("The sorted slice is:", a)
}

// Len returns the length of a 1-based index slice.
func (h *heap) Len() int {
	return len(h.xi) - 1
}

// Heapify turns the receiver into a slice in heap form.
func (h *heap) Heapify() {
	// Prepend a 0 into the 0 index
	h.Prepend(0)
```

```

	for idx := 1; idx < len(h.xi); idx++ {
		h.bubbleUp(idx)
	}
}

// Prepend adds the value, v, to the beginning of a slice.
func (h *heap) Prepend(v int) {
	h.xi = append(h.xi, v)
	copy(h.xi[1:], h.xi)
	h.xi[0] = v
}

// Insert the given key, k, into the heap
func (h *heap) Insert(k int) {
	h.xi = append(h.xi, k)
	h.bubbleUp(h.Len())
}

// bubbleUp moves the key at index, k, into position in the heap.
func (h *heap) bubbleUp(k int) {
	p, ok := parentIndex(k)
	if !ok {
		return // k is the root node
	}
	if h.xi[p] > h.xi[k] {
		h.swap(k, p)
		h.bubbleUp(p)
	}
}

func (h *heap) swap(a, b int) {
	h.xi[a], h.xi[b] = h.xi[b], h.xi[a]
}

// Return the index of the parent of the node at index k.
func parentIndex(k int) (int, bool) {
	// k is the root node
	if k < 2 {
		return 0, false
	}
	return k / 2, true
}

// Return the index of the left child for the parent node at index k.
func (h *heap) leftIndex(k int) (int, bool) {
```

```go
	c := 2 * k
	if c > h.Len() || k == 0 {
		return 0, false
	}
	return c, true
}

// Return the index of the right child for the parent node at index k.
func (h *heap) rightIndex(k int) (int, bool) {
	c := 2*k + 1
	if c > h.Len() || k == 0 {
		return 0, false
	}
	return c, true
}

// ExtractMin returns the minimum value of the heap, the root.
func (h *heap) ExtractMin() (int, bool) {
	// Check if the heap only has the 0-element
	if h.Len() == 0 {
		return 0, false
	}
	root := h.xi[1]
	// Swap the value in the last index position into the root position.
	h.xi[1] = h.xi[h.Len()]
	h.xi = h.xi[:h.Len()]
	h.bubbleDown(1)
	return root, true
}

func (h *heap) bubbleDown(idx int) {
	min := idx
	// Find the smallest value between idx and the two children.
	left, ok := h.leftIndex(idx)
	if ok {
		if h.xi[min] > h.xi[left] {
			min = left
		}
	}
	r, ok := h.rightIndex(idx)
	if ok {
		if h.xi[min] > h.xi[r] {
			min = r
		}
	}
	if min != idx {
```

```
            h.swap(idx, min)
            h.bubbleDown(min)
        }
}
```