COMP2521 24T1

Priority Queues and Heaps

Kevin Luxa

cs2521@cse.unsw.edu.au

priority queues
binary heaps
heap sort

Motivation

We have learned about types of collections where items are inserted and then deleted based on insertion order

stack
last in, first out

queue
first in, first out

Motivation

There are applications where we want to process items based on priority

Examples:
Huffman coding
Dijkstra's algorithm
Prim's algorithm

Priority Queues

A priority queue is an abstract data type
where each item has an associated priority.

It supports the following operations:

insert
insert an item with an associated priority

delete
delete (and return) the item with the highest priority

peek
get the item with the highest priority, without deleting it

is empty
check if the priority queue is empty

Priority is often given by an integer value.

Depending on the application,
either a large priority value or small priority value
could be taken to mean "high priority".

Here we'll take a larger priority value to mean higher priority.

Priority Queues

Interface

```c
typedef struct pq *Pq;

/** Creates a new, empty pq */
Pq PqNew(void);

/** Frees memory allocated to a pq */
void PqFree(Pq pq);

/** Adds an item with priority to a pq */
void PqInsert(Pq pq, Item item, int priority);

/** Deletes and returns the item with the highest priority */
Item PqDelete(Pq pq);

/** Returns the item with the highest priority */
Item PqPeek(Pq pq);

/** Returns true if the pq is empty, false otherwise */
bool PqIsEmpty(Pq pq);
```

Priority Queues

Example Usage

```c
Pq pq = PqNew();

PqInsert(pq, "alice", 4);
PqInsert(pq, "bob", 3);
PqInsert(pq, "andrew", 30);
PqInsert(pq, "jas", 35);

printf("%s\n", PqDelete(pq)); // jas
printf("%s\n", PqDelete(pq)); // andrew

PqInsert(pq, "jake", 23);
PqInsert(pq, "sasha", 25);

printf("%s\n", PqPeek(pq));   // sasha
printf("%s\n", PqDelete(pq)); // sasha
printf("%s\n", PqDelete(pq)); // jake
printf("%s\n", PqDelete(pq)); // alice
printf("%s\n", PqDelete(pq)); // bob

if (PqIsEmpty(pq)) {
    printf("the queue is empty\n");
}

PqFree(pq);
```

Priority Queues

How to implement a priority queue?

unordered array

ordered array

linked list (unordered/ordered)

Priority Queue

Unordered array implementation

unordered array

[0]	[1]	[2]	[3]	[4]	[5]
alice 4	bob 3	andrew 30	jas 35	jake 23	sasha 25

Performance?
Insert: $O(1)$
Delete: $O(n)$
Peek: $O(n)$
Is empty: $O(1)$

Priority Queue

Ordered array implementation

ordered array

[0]	[1]	[2]	[3]	[4]	[5]
bob 3	alice 4	jake 23	sasha 25	andrew 30	jas 35

Performance?

Insert: $O(n)$

Delete: $O(1)$

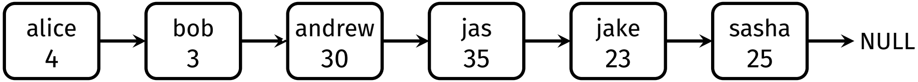
Peek: $O(1)$

Is empty: $O(1)$

Priority Queue

Unordered linked list implementation

unordered linked list



Performance?

Insert: $O(1)$

Delete: $O(n)$

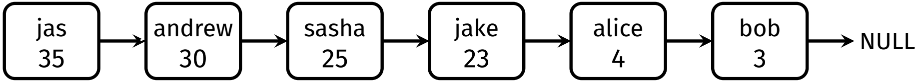
Peek: $O(n)$

Is empty: $O(1)$

Priority Queue

Ordered linked list implementation

ordered linked list



Performance?

Insert: $O(n)$

Delete: $O(1)$

Peek: $O(1)$

Is empty: $O(1)$

Summary (so far)

Data Structure	Insert	Delete	Peek	Is Empty
Unordered array	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ordered array	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Unordered linked list	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ordered linked list	$O(n)$	$O(1)$	$O(1)$	$O(1)$

Heaps

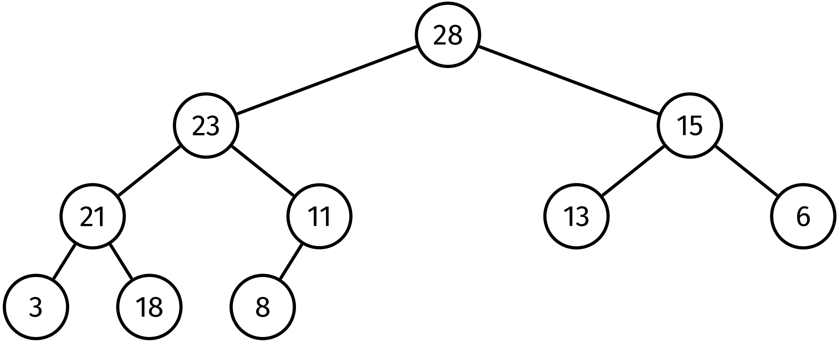
A heap is a tree-based data structure which satisfies the heap property.

The heap property specifies how values in the heap should be ordered, and depends on the kind of heap:

In a max heap, the value in each node must be greater than or equal to the values in its children.

In a min heap, the value in each node must be less than or equal to the values in its children.

Example max heap:



In this lecture we will focus on *max heaps*
(min heaps can be implemented very similarly)

Heaps

There are many variants of heaps,
for example:

binary heap, binomial heap, Fibonacci heap,
leftist heap, pairing heap, soft heap,
...

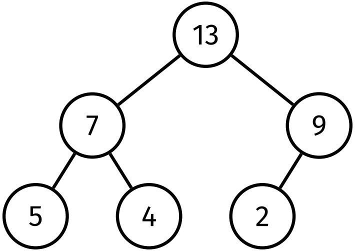
We will consider just the binary heap.

A binary heap is a heap that
takes the form of a binary tree,
and satisfies the following properties:

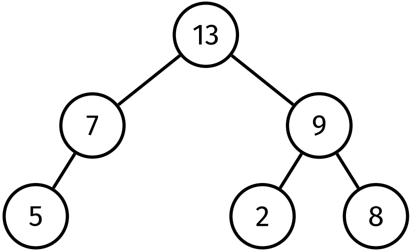
heap property
as defined above

completeness property
all levels of the tree (except possibly the last) must be fully filled
and the last level must be filled from left to right

Binary Heaps



satisfies heap property
satisfies completeness
$\Rightarrow$ is a binary heap

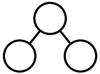
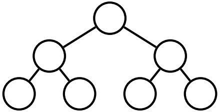


satisfies heap property
does *not* satisfy completeness
$\Rightarrow$ is *not* a binary heap

Binary Heaps

A result of the completeness property
is that binary heaps always contain $\lfloor \log_2 n \rfloor + 1$ levels
where n is the number of nodes.

This will be relevant for analysis.

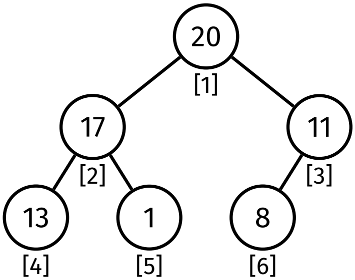
n	number of levels	heap
1	1	
2-3	2	
4-7	3	
...	...	...

Binary Heaps

...as arrays

Heaps are usually implemented with an array.

For a binary heap,
index 1 of the array contains the root item,
the next two indices contain the root's children,
the next four indices contain the children of the root's children,
and so on.

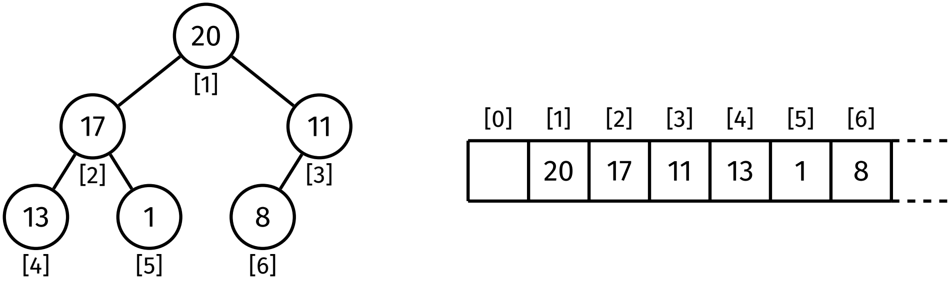


[0]	[1]	[2]	[3]	[4]	[5]	[6]
	20	17	11	13	1	8

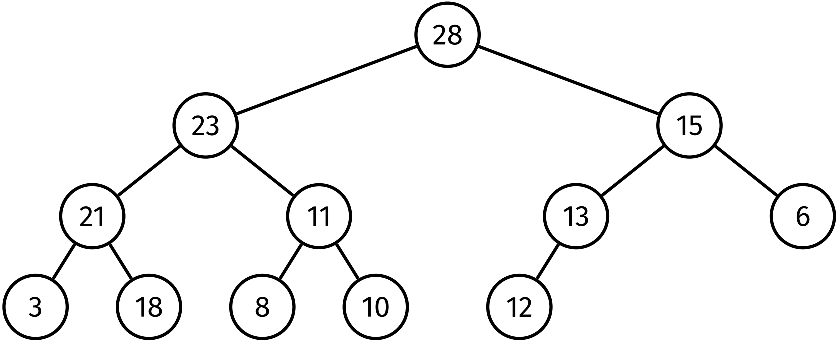
This arrangement gives rise to a useful property:

- For an item at index i:
  - Its left child is located at index $2i$
  - Its right child is located at index $2i + 1$
  - Its parent is located at index $\lfloor i/2 \rfloor$

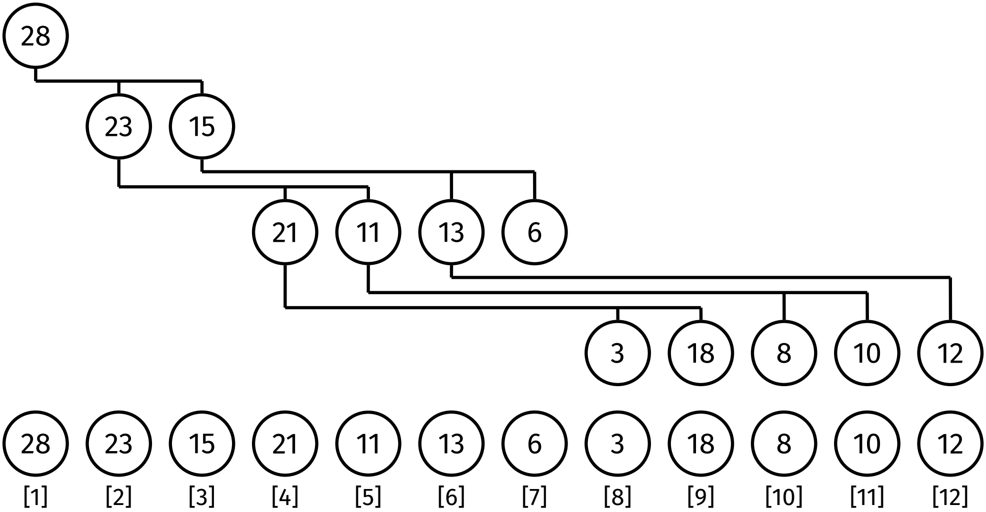
This makes it efficient to move "up" and "down" the tree.



Consider this max heap:



The heap as an array:



Assuming integer items:

```
struct heap {
    int *items;
    int  numItems;
    int  capacity;
};
```

```
struct heap *heapNew(void) {
    struct heap *heap = malloc(sizeof(struct heap));

    heap->numItems = 0;
    heap->capacity = INITIAL_CAPACITY;
    heap->items = malloc((heap->capacity + 1) * sizeof(int));

    return heap;
}
```

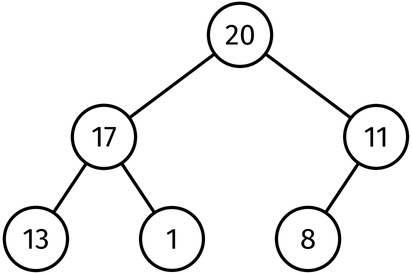
Binary Heap Insertion

Insertion is a two-step process:

1. Add new item at next available position on bottom level
   i.e., after the last item
   - New item may violate the heap property
2. **Fix up**: While new item is greater than its parent (and not at the root), swap with its parent
   - This re-organises items along the path to the root and restores the heap property

Binary Heap Insertion

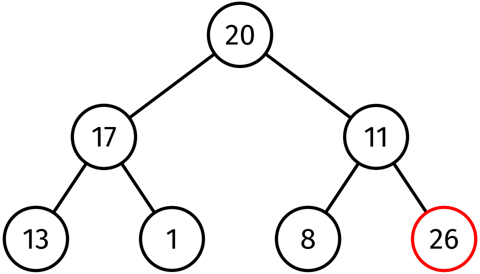
Example: Insert 26



Binary Heap Insertion

Example: Insert 26

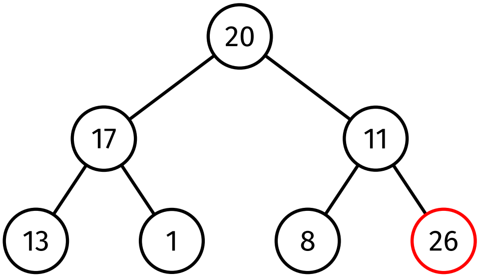
Insert 26 after the last item (8)



Binary Heap Insertion

Example: Insert 26

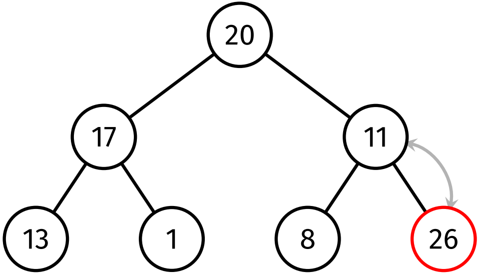
Fix up



Binary Heap Insertion

Example: Insert 26

Fix up
26 is greater than its parent (11) $\Rightarrow$ swap

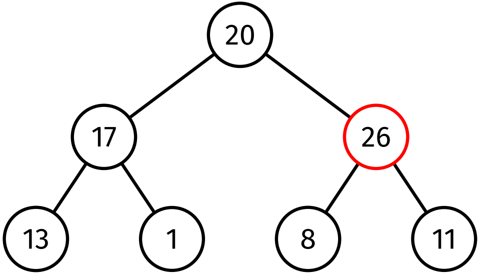


Binary Heap Insertion

Example: Insert 26

Fix up

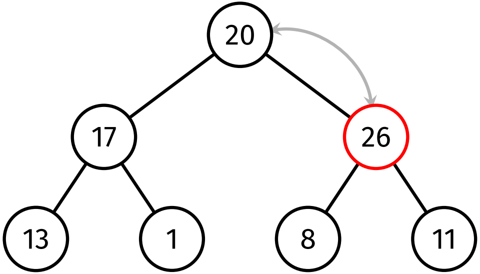
26 is greater than its parent (11) $\Rightarrow$ swap



Binary Heap Insertion

Example: Insert 26

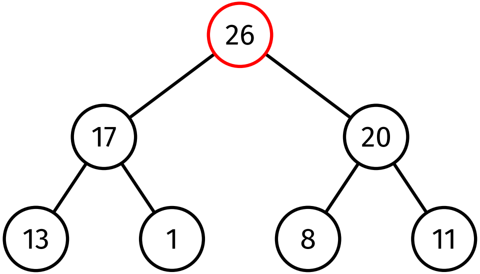
Fix up
26 is greater than its parent (20) $\Rightarrow$ swap



Binary Heap Insertion

Example: Insert 26

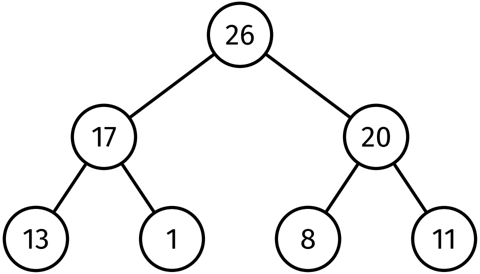
Fix up
26 is greater than its parent (20) $\Rightarrow$ swap



Binary Heap Insertion

Example: Insert 26

Done



Binary Heap Insertion

Example

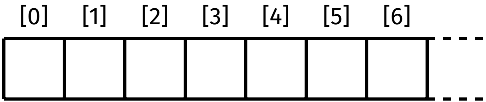
Insert the following items into an initially empty max heap:

17 25 8 6 30 13

[0]	[1]	[2]	[3]	[4]	[5]	[6]	...

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

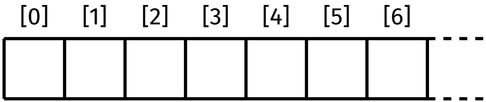


Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



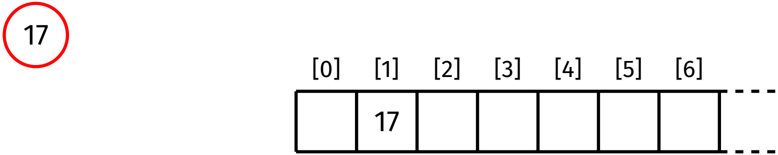
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 17 to the heap



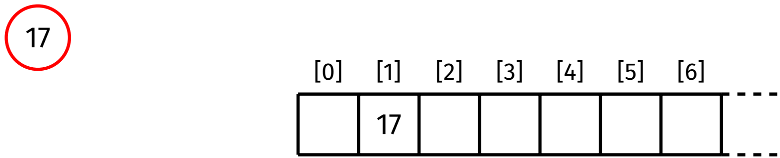
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

17 is at the root - done



Insert the following items into an initially empty max heap:

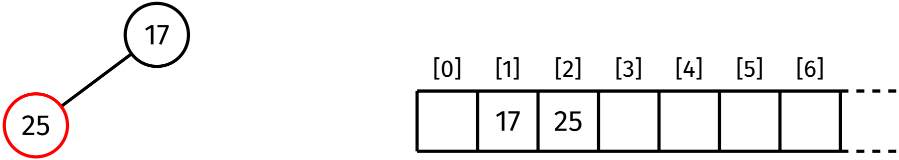
17 25 8 6 30 13

[0]	[1]	[2]	[3]	[4]	[5]	[6]
	17					

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 25 after the last item



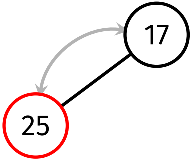
Binary Heap Insertion

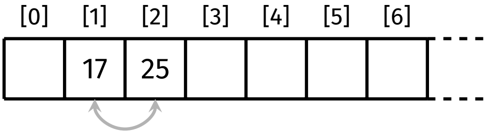
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

25 is greater than its parent (17) - swap

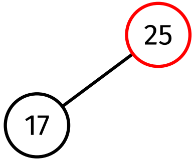




Insert the following items into an initially empty max heap:

17 25 8 6 30 13

25 is greater than its parent (17) - swap



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17				

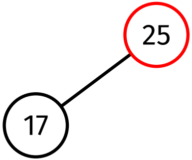
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

25 is at the root - done

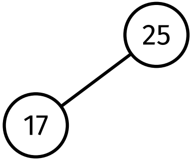


[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17				

Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17				

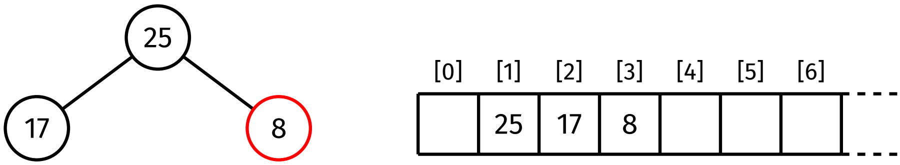
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 8 after the last item



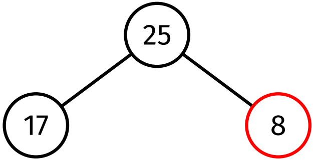
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

8 is not greater than its parent (25) - done



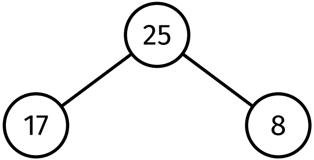
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	8			

Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	8			

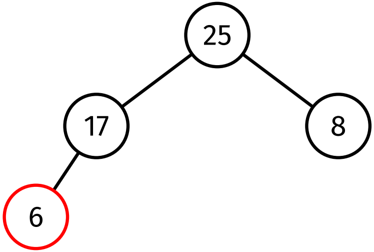
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 6 after the last item



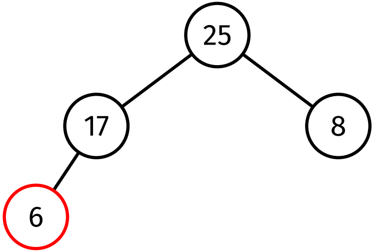
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	8	6		

Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

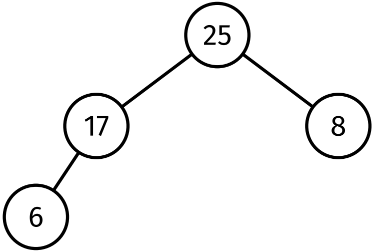
6 is not greater than its parent (17) - done



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	8	6		

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	8	6		

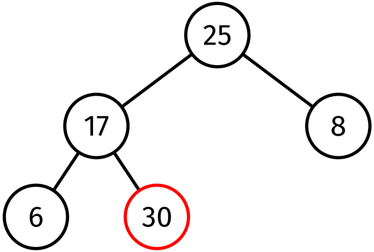
Binary Heap Insertion

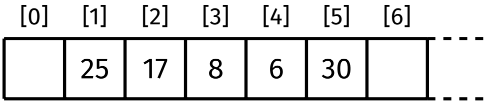
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 30 after the last item





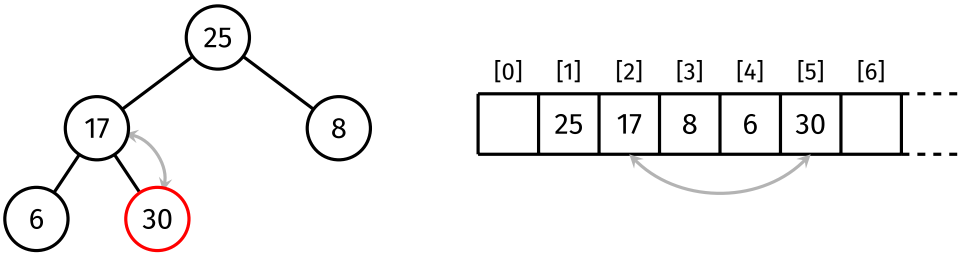
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

30 is greater than its parent (17) - swap



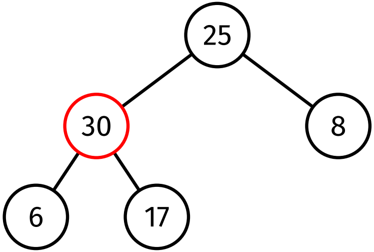
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

30 is greater than its parent (17) - swap



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	30	8	6	17	

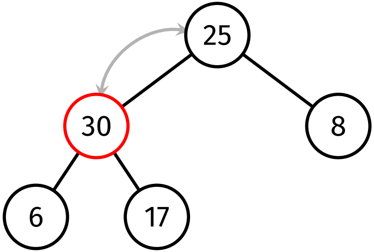
Binary Heap Insertion

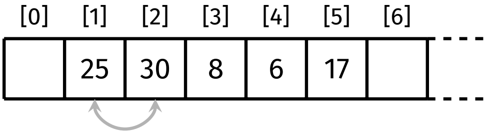
Example

Insert the following items into an initially empty max heap:

17 25 8 6 **30** 13

30 is greater than its parent (25) - swap

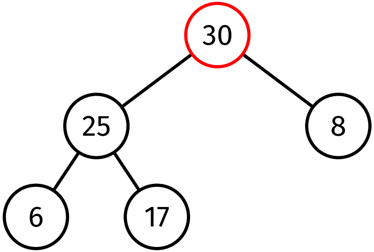




Insert the following items into an initially empty max heap:

17 25 8 6 30 13

30 is greater than its parent (25) - swap



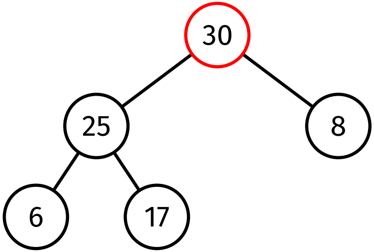
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	8	6	17	

Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

30 is at the root - done

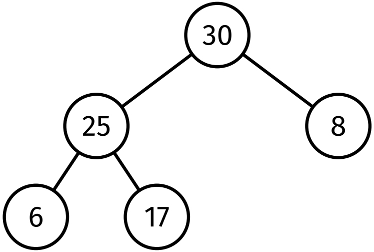


[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	8	6	17	

Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	8	6	17	

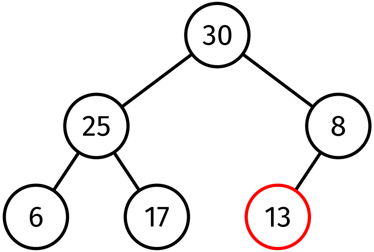
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

Add 13 after the last item



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	8	6	17	13

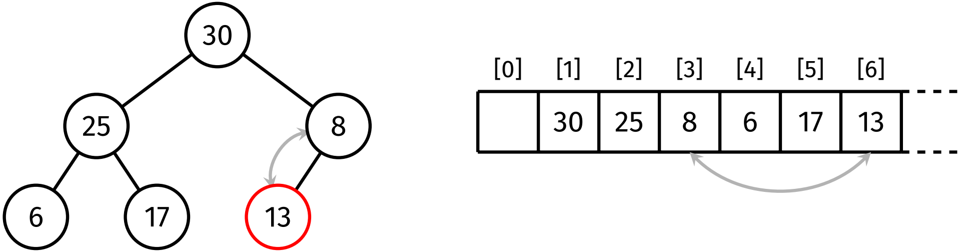
Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

13 is greater than its parent (8) - swap

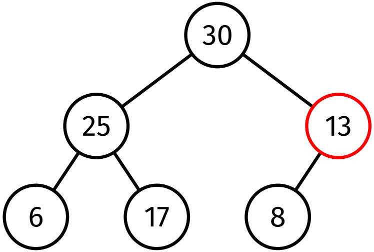


Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

13 is greater than its parent (8) - swap



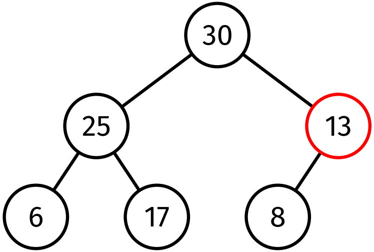
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	13	6	17	8

Binary Heap Insertion
Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13

13 is not greater than its parent (30) - done



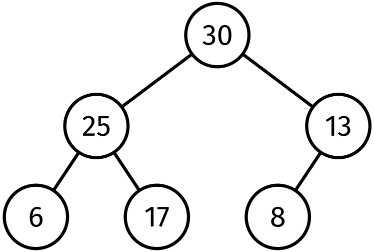
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	13	6	17	8

Binary Heap Insertion

Example

Insert the following items into an initially empty max heap:

17 25 8 6 30 13



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	13	6	17	8

Binary Heap Insertion

Implementation


```
void heapInsert(struct heap *heap, Item it) {
    if (heap->numItems == heap->capacity) {
        // resize
    }
    heap->numItems++;
    heap->items[heap->numItems] = it;
    fixUp(heap->items, heap->numItems);
}

void fixUp(Item items[], int i) {
    // while index i is not the root and
    // item at index i is greater than its parent
    while (i > 1 && items[i] > items[i / 2]) {
        swap(items, i, i / 2);
        i = i / 2;
    }
}
```

Cost of insertion:

- Add new item after last item $\Rightarrow O(1)$
- Fix up considers one item on each level in the worst case
- Heap is a complete tree $\Rightarrow O(\log n)$ levels
- Therefore, worst-case time complexity is $O(\log n)$

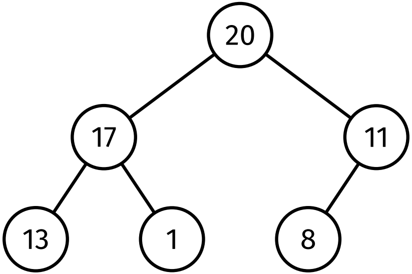
Binary Heap Deletion

Deletion is a three-step process:

1. Replace root item with last item
   - Last item = bottom-most, rightmost item
   - Let this item be i
2. Remove last item
3. Fix down: While i is less than its greater child, swap it with its greater child
   - This restores the heap property

Binary Heap Deletion

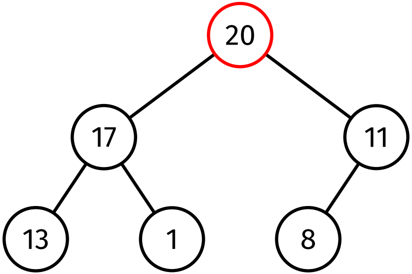
Example: Delete from this max heap



Binary Heap Deletion

Example: Delete from this max heap

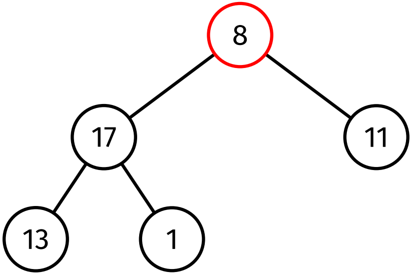
Delete 20, replace with 8



Binary Heap Deletion

Example: Delete from this max heap

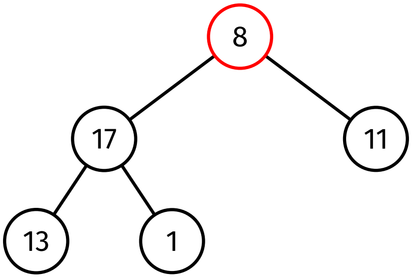
Delete 20, replace with 8



Binary Heap Deletion

Example: Delete from this max heap

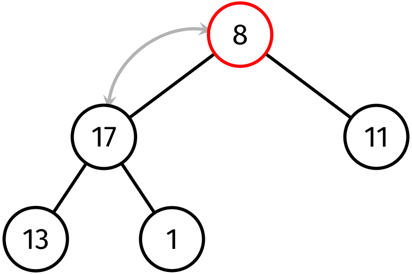
Fix down



Binary Heap Deletion

Example: Delete from this max heap

Fix down
8 is less than its greater child (17) $\Rightarrow$ swap

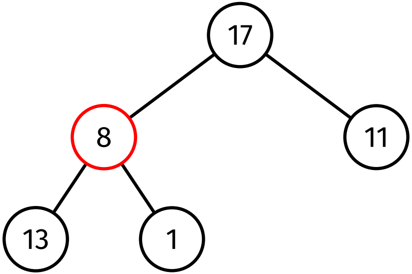


Binary Heap Deletion

Example: Delete from this max heap

Fix down

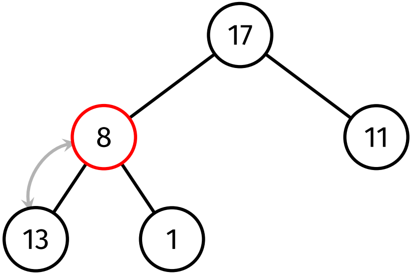
8 is less than its greater child (17) $\Rightarrow$ swap



Binary Heap Deletion

Example: Delete from this max heap

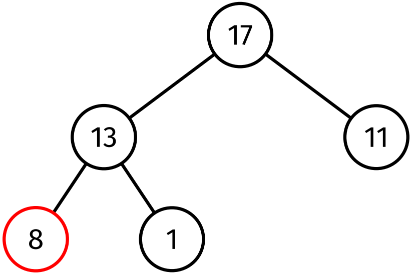
Fix down
8 is less than its greater child (13) $\Rightarrow$ swap



Binary Heap Deletion

Example: Delete from this max heap

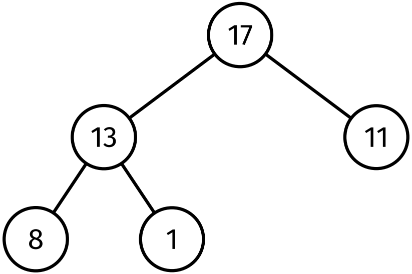
Fix down
8 is less than its greater child (13) $\Rightarrow$ swap



Binary Heap Deletion

Example: Delete from this max heap

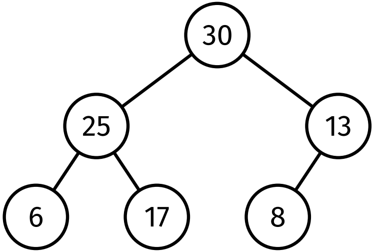
Done



Binary Heap Deletion

Example

Delete from the following max heap until it is empty:



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	13	6	17	8

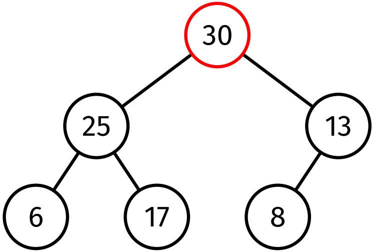
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30

Deleting 30



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	30	25	13	6	17	8

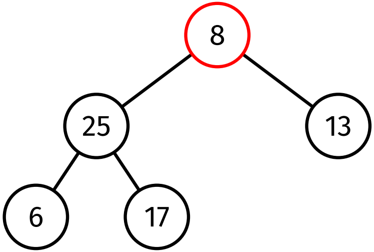
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30

Replace 30 with last item (8)

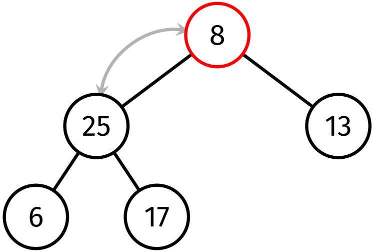


[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	25	13	6	17	

Delete from the following max heap until it is empty:

30

8 is less than its greater child (25) - swap



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	25	13	6	17	

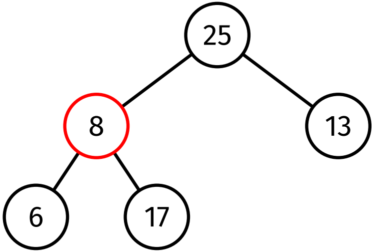
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30

8 is less than its greater child (25) - swap



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	8	13	6	17	

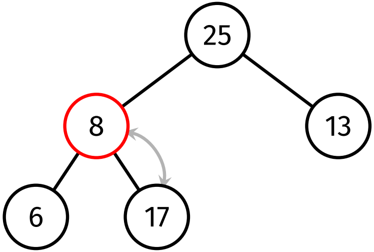
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30

8 is less than its greater child (17) - swap



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	8	13	6	17	

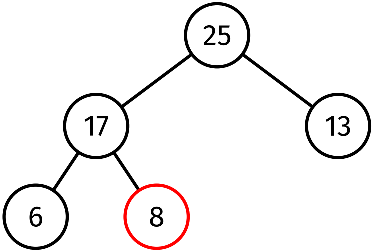
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30

8 is less than its greater child (17) - swap



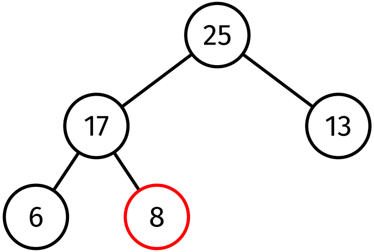
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	13	6	8	

Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30

8 is at a leaf - done



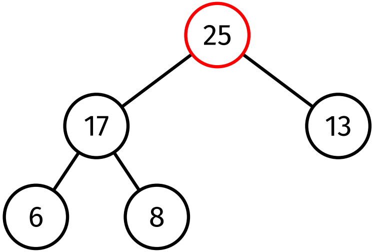
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	13	6	8	

Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25

Deleting 25



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	25	17	13	6	8	

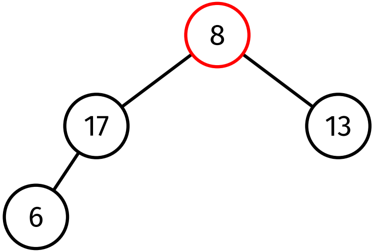
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25

Replace 25 with last item (8)



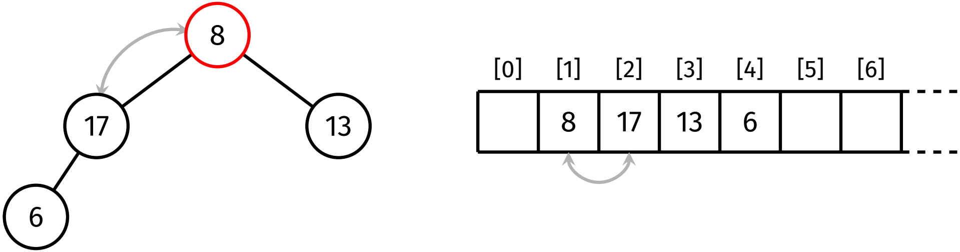
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	17	13	6		

Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25

8 is less than its greater child (17) - swap



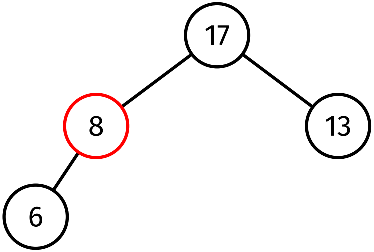
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25

8 is less than its greater child (17) - swap



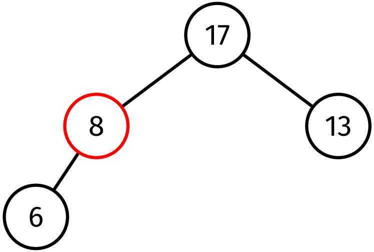
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	17	8	13	6		

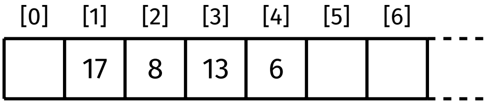
Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25

8 is not less than its greater child (6) - done





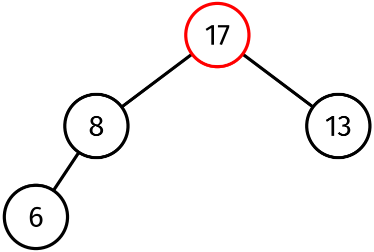
Binary Heap Deletion

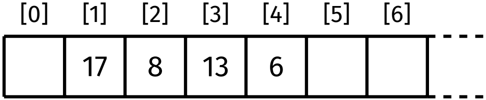
Example

Delete from the following max heap until it is empty:

30 25 17

Deleting 17





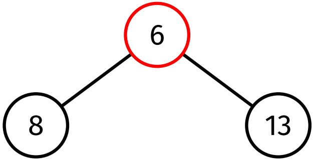
Binary Heap Deletion

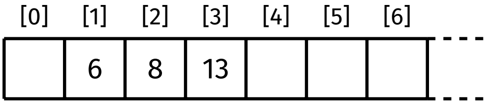
Example

Delete from the following max heap until it is empty:

30 25 17

Replace 17 with last item (6)

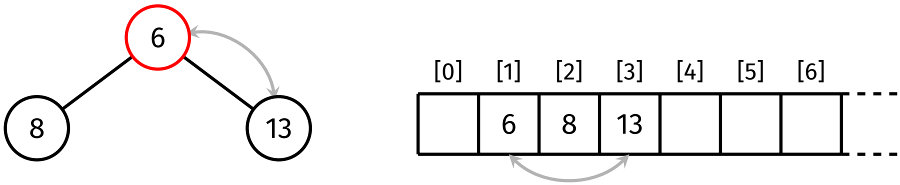




Delete from the following max heap until it is empty:

30 25 17

6 is less than its greater child (13) - swap



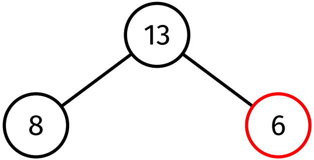
Binary Heap Deletion

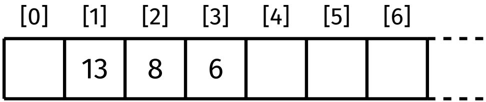
Example

Delete from the following max heap until it is empty:

30 25 17

6 is less than its greater child (13) - swap



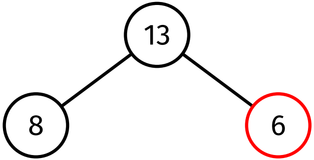


Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25 17

6 is at a leaf - done



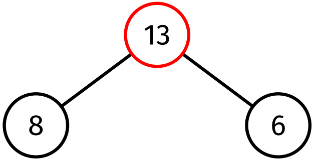
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	13	8	6			

Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25 17 13

Deleting 13

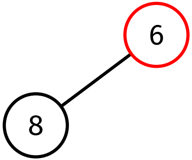


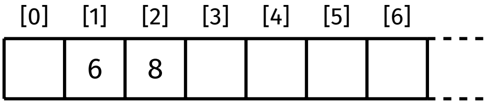
[0]	[1]	[2]	[3]	[4]	[5]	[6]
	13	8	6			

Delete from the following max heap until it is empty:

30 25 17 13

Replace 13 with last item (6)



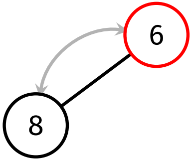


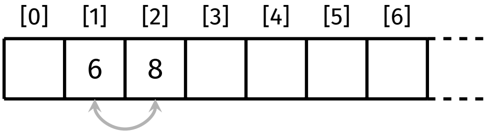
Binary Heap Deletion
Example

Delete from the following max heap until it is empty:

30 25 17 13

6 is less than its greater child (8) - swap

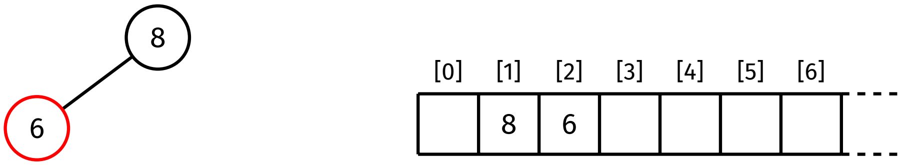




Delete from the following max heap until it is empty:

30 25 17 13

6 is less than its greater child (8) - swap



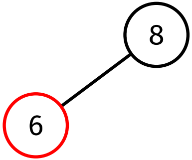
Binary Heap Deletion

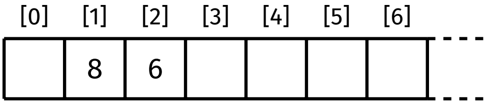
Example

Delete from the following max heap until it is empty:

30 25 17 13

6 is at a leaf - done





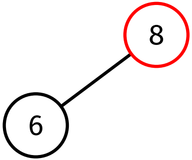
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8

Deleting 8



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	6				

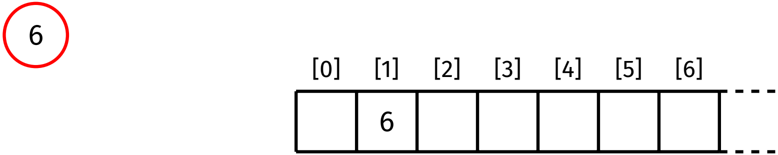
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8

Replace 8 with last item (6)



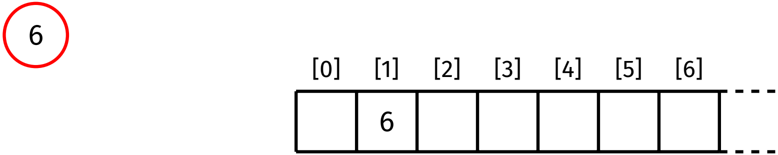
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8

6 is at a leaf - done



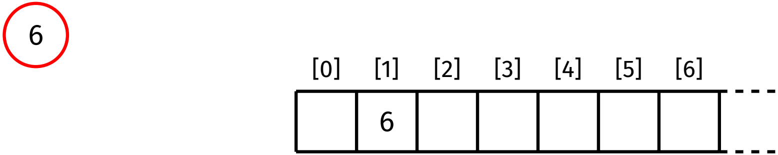
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8 6

Deleting 6



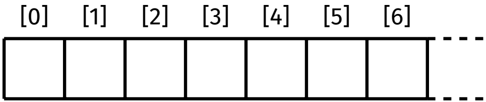
Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8 6

Delete 6



Binary Heap Deletion

Example

Delete from the following max heap until it is empty:

30 25 17 13 8 6

Heap is now empty

[0]	[1]	[2]	[3]	[4]	[5]	[6]

Binary Heap Deletion

Implementation (I)


```
Item heapDelete(struct heap *heap) {
    Item item = heap->items[1];
    heap->items[1] = heap->items[heap->numItems];
    heap->numItems--;
    fixDown(heap->items, 1, heap->numItems);
    return item;
}
```

Binary Heap Deletion

Implementation (II)


```
void fixDown(Item items[], int i, int N) {
    // while index i has at least one child
    while (2 * i <= N) {
        // let j be the index of index i's left child
        int j = 2 * i;

        // if index i's right child is greater than its left child
        if (j < N && items[j] < items[j + 1]) j++;

        // if the item at index i is greater than or equal to both children
        if (items[i] >= items[j]) break;

        swap(items, i, j);

        // move one level down the heap
        i = j;
    }
}
```

Binary Heap Deletion

Analysis

Cost of deletion:

- Replace root by item at end of array $\Rightarrow O(1)$
- Fix down considers two items on each level in the worst case
- Heap is a complete tree $\Rightarrow O(\log n)$ levels
- Therefore, worst-case time complexity is $O(\log n)$

PQ Implementation

Concrete data structures

```
struct pq {
    struct pqItem *items; // array of items
    int numItems;         // number of items stored
    int capacity;         // max number of items
};

struct pqItem {
    Item item;
    int priority;
};
```

PQ Implementation

Constructor

```
Pq PqNew(void) {
    Pq pq = malloc(sizeof(struct pq));

    pq->numItems = 0;
    pq->capacity = INITIAL_CAPACITY;
    pq->items = malloc((pq->capacity + 1) * sizeof(struct pqItem));
    return pq;
}
```

PQ Implementation

Insertion

```c
void PqInsert(Pq pq, Item it, int priority) {
    if (pq->numItems == pq->capacity) {
        // resize array
    }

    pq->numItems++;
    pq->items[pq->numItems] = (struct pqItem){it, priority};
    fixUp(pq->items, pq->numItems);
}

void fixUp(struct pqItem items[], int i) {
    while (i > 1 && items[i].priority > items[i / 2].priority) {
        swap(items, i, i / 2);
        i = i / 2;
    }
}
```

PQ Implementation

Deletion

```
Item PqDelete(Pq pq) {
    Item item = pq->items[1].item;
    pq->items[1] = pq->items[pq->numItems];
    pq->numItems--;
    fixDown(pq->items, 1, pq->numItems);
    return item;
}

void fixDown(struct pqItem items[], int i, int N) {
    while (2 * i <= N) {
        int j = 2 * i;
        if (j < N && items[j].priority < items[j + 1].priority) j++;
        if (items[i].priority >= items[j].priority) break;
        swap(items, i, j);
        i = j;
    }
}
```

Data Structure	Insert	Delete	Peek	Is Empty
Unordered array	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ordered array	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Unordered linked list	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ordered linked list	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$

Heap Sort

Heap sort is a sorting algorithm that uses a heap!

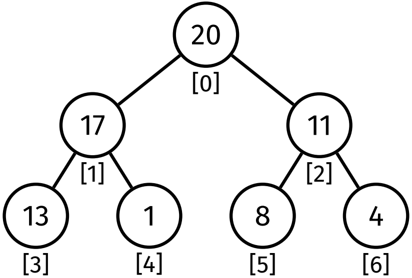
Method:

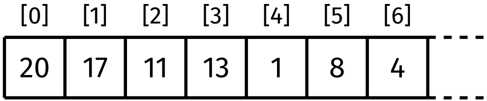
- Build up a heap within the original array
  - This is called “heapify”
- Repeatedly delete from the heap
  - Each time an element is deleted, place it at the end of the heap

Heap Sort

Adjusted indexing scheme:

- For an item at index i:
  - Its children are at indices $2i + 1$ and $2i + 2$
  - Its parent is located at index $\lfloor (i - 1)/2 \rfloor$





How to build up a heap within the original array?

Idea:

Use a similar idea to insertion sort!

Take first element and treat as a heap of size 1

Take next element and insert into the heap, which increases the size of the heap by one

Repeat for remaining elements

Example:

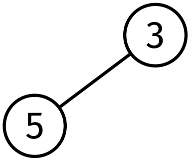
3	5	1	6	7	2	4

Take first element and treat as heap of size 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	5	1	6	7	2	4

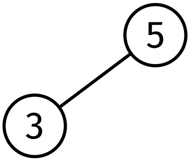
Insert 5 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	5	1	6	7	2	4



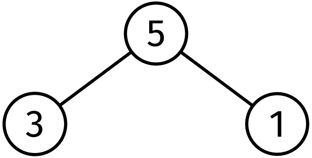
Insert 5 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
5	3	1	6	7	2	4



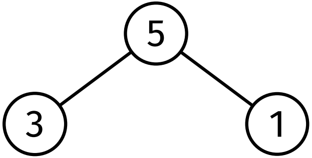
Insert 1 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
5	3	1	6	7	2	4

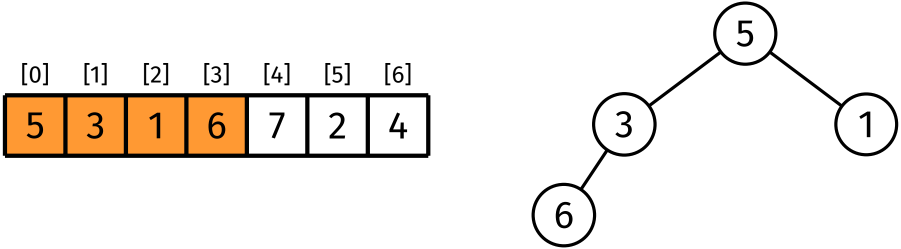


Insert 1 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
5	3	1	6	7	2	4



Insert 6 into the heap

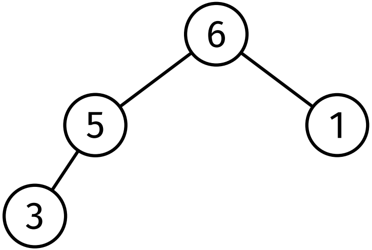


Heap Sort

Heapify, Fix Up Method - Example

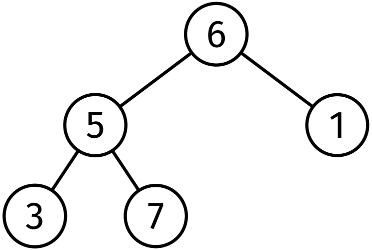
Insert 6 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	5	1	3	7	2	4



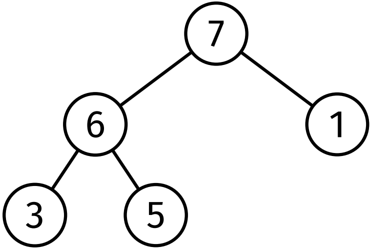
Insert 7 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	5	1	3	7	2	4



Insert 7 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	1	3	5	2	4

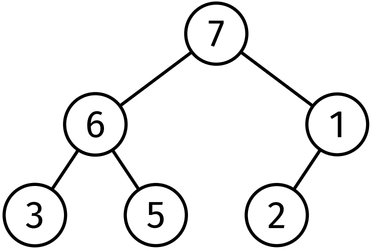


Heap Sort

Heapify, Fix Up Method - Example

Insert 2 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	1	3	5	2	4

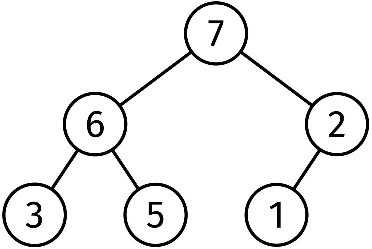


Heap Sort

Heapify, Fix Up Method - Example

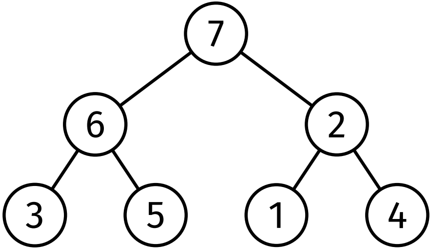
Insert 2 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	2	3	5	1	4



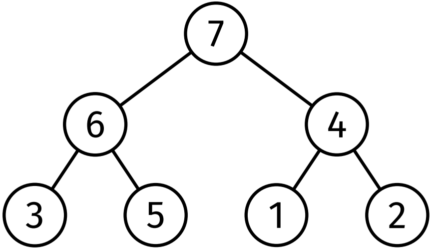
Insert 4 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	2	3	5	1	4



Insert 4 into the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	4	3	5	1	2



Heap Sort

Heapify, Fix Up Method - Implementation


```
void heapify(Item items[], int size) {
    for (int i = 1; i < size; i++) {
        fixUp(items, i);
    }
}

void fixUp(Item items[], int i) {
    while (i > 0 && items[i] > items[(i - 1) / 2]) {
        swap(items, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}
```

Analysis:

- Inserting into a heap is $O(\log n)$
- Therefore, inserting n items into an initially empty heap is $O(\log 1 + \log 2 + \log 3 + \ldots + \log n) = O(\log n!) = O(n \log n)$

Heap Sort

Heapify, Fix Down Method

Heapify can be implemented more efficiently by performing a **fix down** on every element in the *first half* of the array in reverse (i.e., from right to left)

Heap Sort

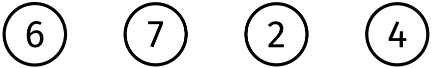
Heapify, Fix Down Method - Example

Example:

3	5	1	6	7	2	4

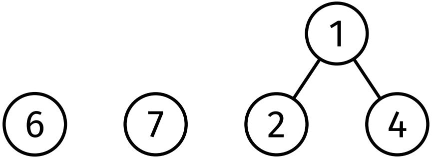
Treat each element in the second half of the array as a heap of size 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	5	1	6	7	2	4



Perform fix down on 1

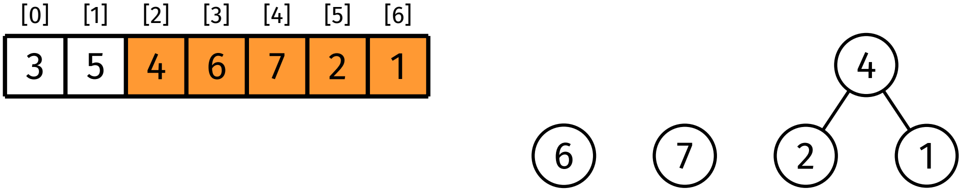
[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	5	1	6	7	2	4



Heap Sort

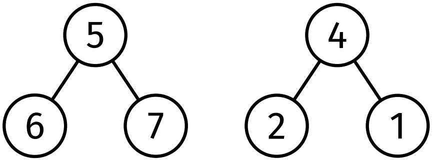
Heapify, Fix Down Method - Example

Perform fix down on 1



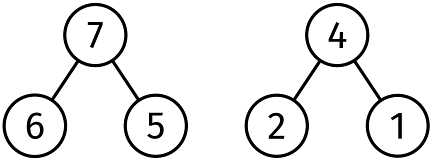
Perform fix down on 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	5	4	6	7	2	1



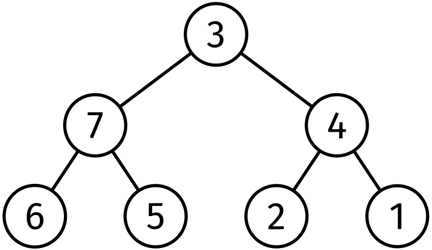
Perform fix down on 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	7	4	6	5	2	1



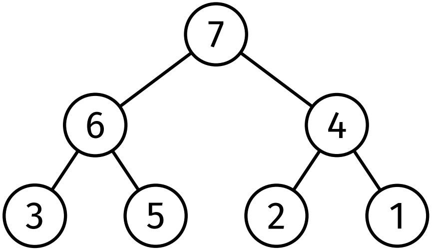
Perform fix down on 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	7	4	6	5	2	1



Perform fix down on 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	4	3	5	2	1



Heap Sort

Heapify, Fix Down Method - Implementation

```
void heapify(Item items[], int size) {
    for (int i = size / 2 - 1; i >= 0; i--) {
        fixDown(items, i, size - 1);
    }
}

void fixDown(Item items[], int i, int N) {
    while (2 * i + 1 <= N) {
        int j = 2 * i + 1;
        if (j < N && items[j] < items[j + 1]) j++;
        if (items[i] >= items[j]) break;
        swap(items, i, j);
        i = j;
    }
}
```

This implementation of heapify is $O(n)$.

Why?

Most of the items in a heap are on the lowest levels.

Heap Sort

De-Heapify

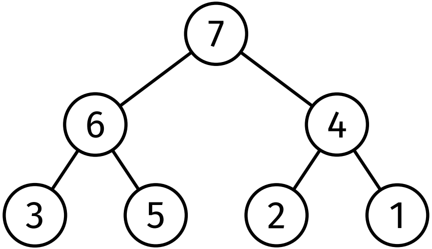
After the array has been heapified, repeatedly delete from the heap, each time placing the deleted item at the end of the heap.

Example:

7	6	4	3	5	2	1

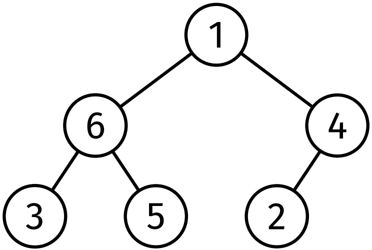
Delete 7 from the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
7	6	4	3	5	2	1



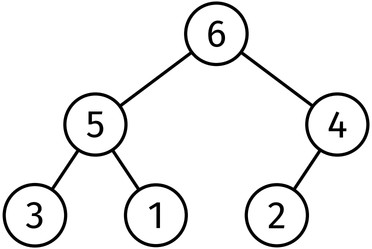
Delete 7 from the heap
Perform fix down on 1 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	6	4	3	5	2	7

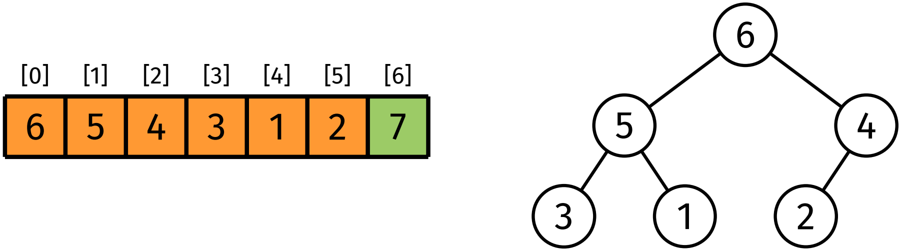


Delete 7 from the heap
Perform fix down on 1 to restore heap

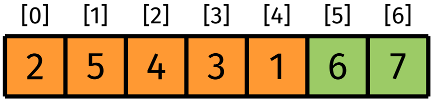
[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	5	4	3	1	2	7

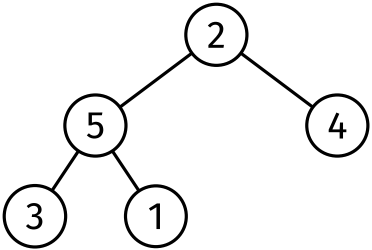


Delete 6 from the heap

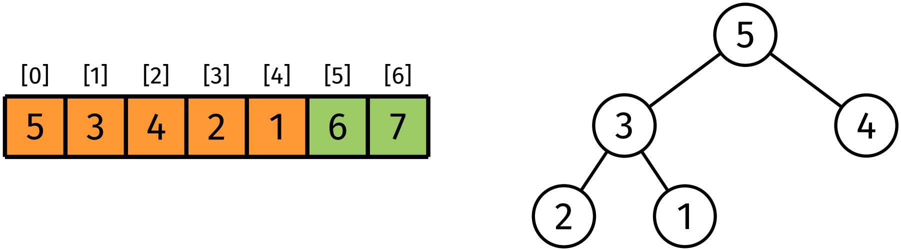


Delete 6 from the heap
Perform fix down on 2 to restore heap



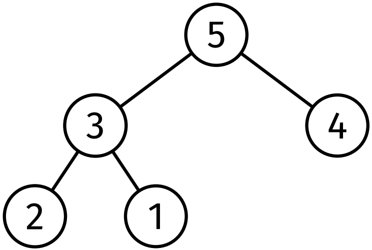


Delete 6 from the heap
Perform fix down on 2 to restore heap



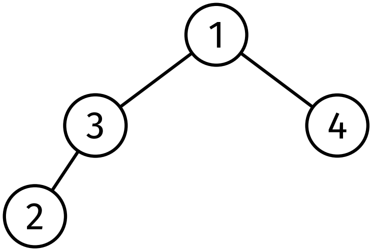
Delete 5 from the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
5	3	4	2	1	6	7



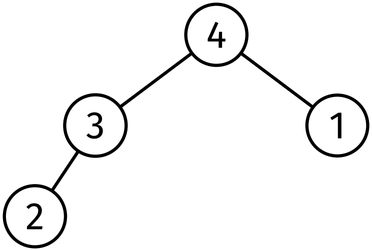
Delete 5 from the heap
Perform fix down on 1 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	4	2	5	6	7



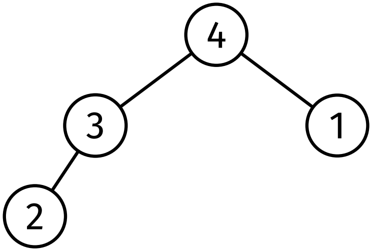
Delete 5 from the heap
Perform fix down on 1 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
4	3	1	2	5	6	7



Delete 4 from the heap

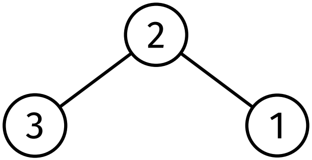
[0]	[1]	[2]	[3]	[4]	[5]	[6]
4	3	1	2	5	6	7



Delete 4 from the heap

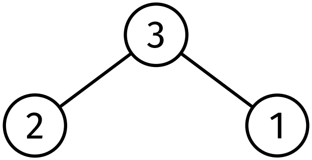
Perform fix down on 2 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
2	3	1	4	5	6	7



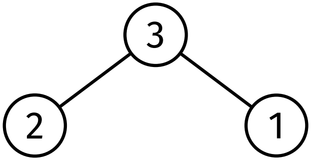
Delete 4 from the heap
Perform fix down on 2 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	2	1	4	5	6	7



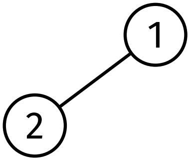
Delete 3 from the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	2	1	4	5	6	7



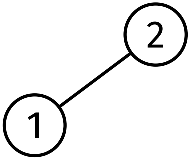
Delete 3 from the heap
Perform fix down on 1 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	2	3	4	5	6	7



Delete 3 from the heap
Perform fix down on 1 to restore heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
2	1	3	4	5	6	7

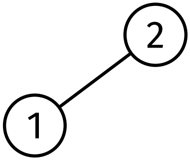


Heap Sort

De-Heapify - Example

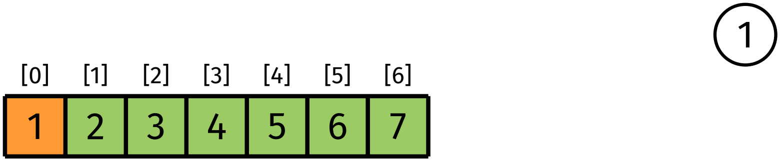
Delete 2 from the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
2	1	3	4	5	6	7

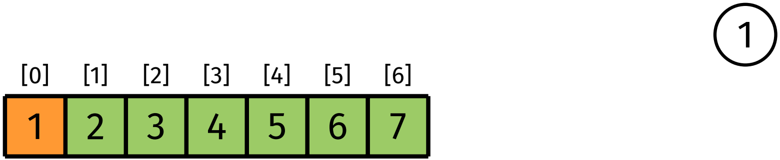


Delete 2 from the heap

Perform fix down on 1 to restore heap



Delete 2 from the heap
Perform fix down on 1 to restore heap



Delete 1 from the heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	2	3	4	5	6	7

Delete 1 from the heap
Done

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	2	3	4	5	6	7

Heap Sort

De-Heapify - Implementation


```
void deheapify(Item items[], int size) {
    while (size > 1) {
        swap(items, 0, size - 1);
        size--;
        fixDown(items, 0, size - 1);
    }
}
```

Analysis:

- Deleting from a heap is $O(\log n)$
- Therefore, deleting all items from a heap of size n is $O(\log n + \log(n-1) + \log(n-2) + \ldots + \log 1) = O(\log n!) = O(n \log n)$

Analysis of heap sort:

- Heapify is $O(n)$
- De-heapify is $O(n \log n)$
- Therefore, heap sort is $O(n \log n)$

Unstable

Due to long-range swaps

Non-adaptive

$O(n \log n)$ on average and if array is sorted

In-place

Sorting is done within original array; does not use temporary arrays

Feedback

https://forms.office.com/r/5c0fb4tvMb