# Programming Abstractions

CS106B

Cynthia Lee

Topics:

- Finish up last topics of Classes
  - › Implement constructor and destructor
  - › (using our ArrayList example)

- Priority Queue ADT
  - › Linked List implementation (sorted vs. unsorted)
  - › Heap data structure implementation
    - • What are binary trees?
    - • What are heaps?
    - • How do we do insert/remove operations on heaps?

Stanford University
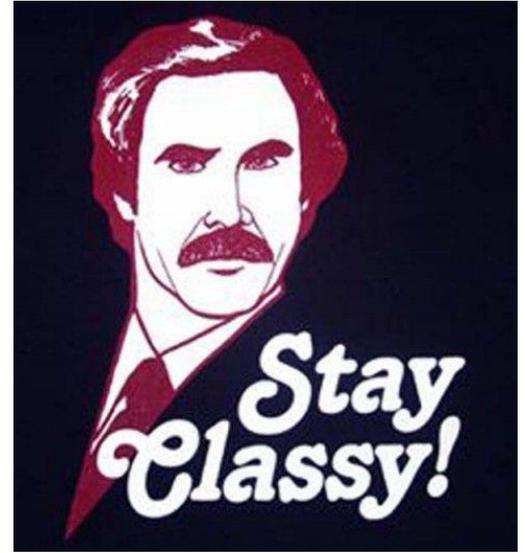
Constructors and Destructors for Classes

A look at what the point of constructors and destructors are, using our ArrayList class as an example

ArrayList.h

```
#ifndef _arraylist_h
#define _arraylist_h
#include <string>
using namespace std;

class ArrayList {
public:
    ArrayList();
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;
    string toString() const;

private:
    int* myElements;    // array of elements
    int myCapacity;     // length of array
    int mySize;         // number of elements added
};
#endif
```

insert solution

```
// in ArrayList.cpp
void ArrayList::insert(int index, int value) {
    // shift right to make room
    for (int i = mySize; i > index; i--) {
        myElements[i] = myElements[i - 1];
    }
    myElements[index] = value;
    mySize++;
}
```

- Here we talk about **myElements** and **mySize** as if they are already set up and ready to go.
- **Question: how do they first get started?**
  - › If **myElements** is on the heap, who called **new** to allocate it?

Constructor solution

```cpp
// in ArrayList.cpp
void ArrayList::ArrayList() {
    myCapacity = 10;// make named constant
    myElements = new int[myCapacity];
    mySize = 0;
}
```

Destructor (12.3)

```
// ClassName.h              // ClassName.cpp
~ClassName();              ClassName::~ClassName() { ...
```

destructor: Called when the object is deleted by the program.

(when the object goes out of {} scope; opposite of a constructor)

- Useful if your object needs to do anything important as it dies:
  › saving any temporary resources inside the object
  › freeing any dynamically allocated memory used by the object's members
  › ...

Destructor solution

```cpp
// in ArrayList.cpp
void ArrayList::~ArrayList() {
    delete[] myElements;
}
```

Priority Queue

Emergency Department waiting room operates as a priority queue: patients are sorted according to priority (urgency), not "first come, first serve" (in computer science, "first in, first out" or FIFO).

Stanford University

Some priority queue implementation options
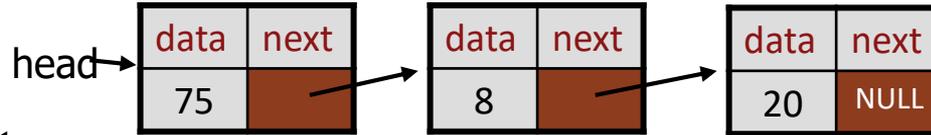
Unsorted linked list

- Insert new element in front
- **Remove by searching list for highest-priority item**
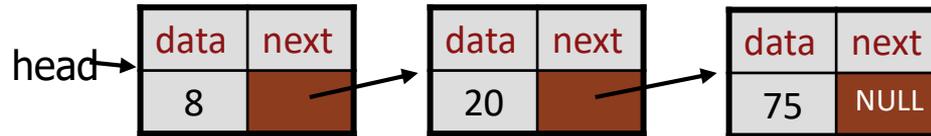
Sorted linked list

- Always insert new elements where they go in priority-sorted order
- **Remove from front (will be highest-priority because sorted)**

Stanford University

Priority queue implementations

Unsorted linked list

head → | data | next |
| 75 | |

→ | data | next |
| 8 | |

→ | data | next |
| 20 | NULL |
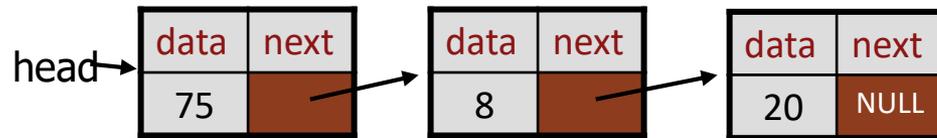
Add is FAST

- Just throw it in the list at the front
- **O(1)**

Remove/peek is SLOW

- Hard to find item the highest priority item—could be anywhere
- **O(N)**

Stanford University

Priority queue implementations

Sorted linked list

data	next
8	

head →

data	next
20	

data	next
75	NULL

Add is SLOW

- Need to step through the list to find where item goes in priority-sorted order
- **O(N)**

Remove/peek is FAST

- Easy to find item you are looking for (first in list)
- **O(1)**

Stanford University

Priority queue implementations

We want the best of both

Fast add AND fast remove/peek

We will investigate trees as a way to get the best of both worlds

Fast add   Fast remove/peek

Binary Heaps

Heap: not to be confused with the Heap!

- The Stack section of memory **is** a Stack like the ADT
- The Heap section of memory **has nothing to do** with the Heap structure.

Stack ADT

Heap data structure

0x0
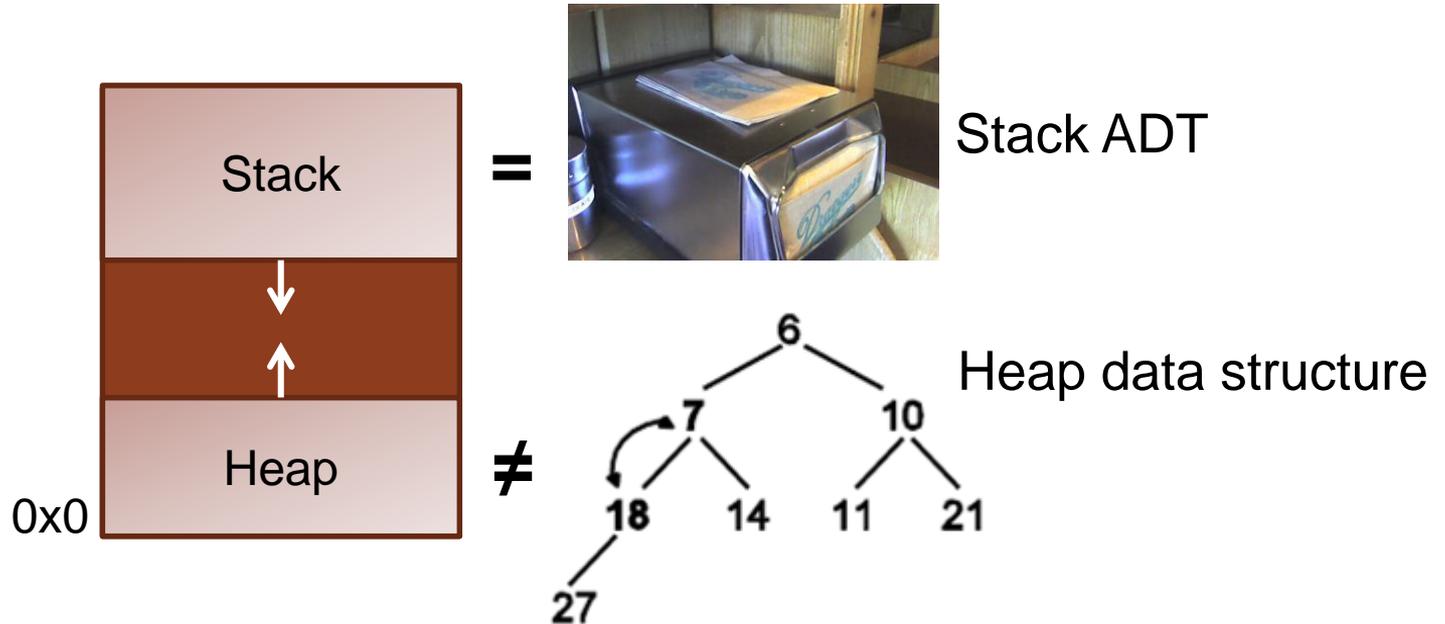
- Probably just happened to reuse the same word

Stanford University

Binary trees

A binary tree

"In computer science, a binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right."" (Thanks, Wikipedia!)

How many of these are valid binary trees?

"In computer science, a binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right."" (Thanks, Wikipedia!)

Stanford University
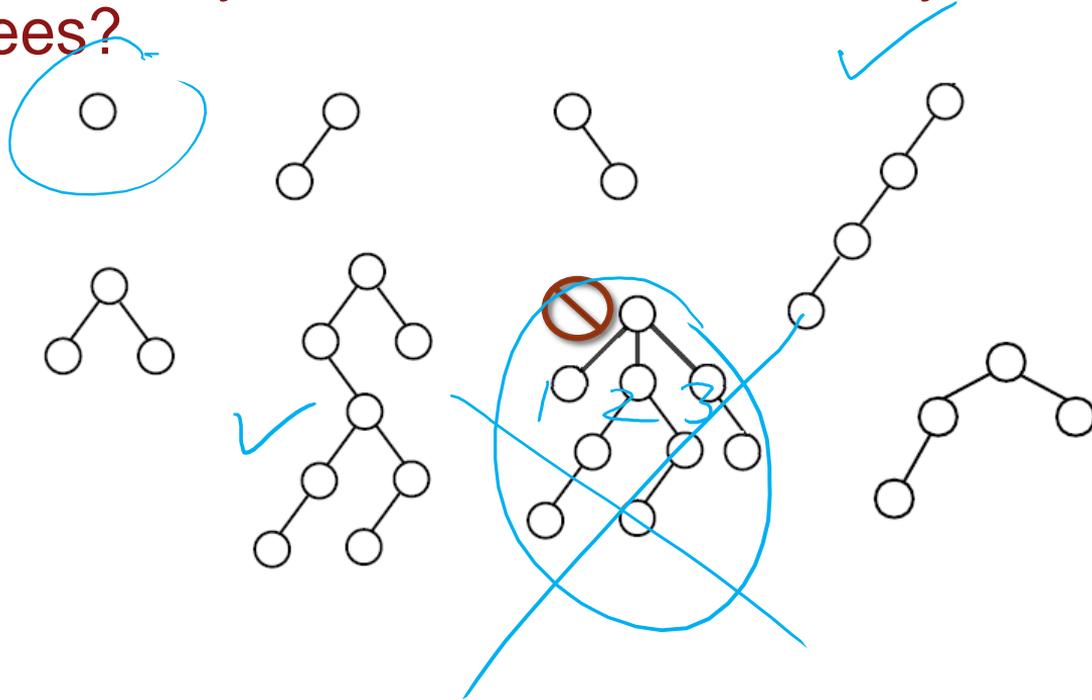
A node struct for binary trees

Similar to a linked list node, it contains **data**, and a pointer to the nearby elements

A binary node tree has two child pointers, **left** and **right**

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};
```
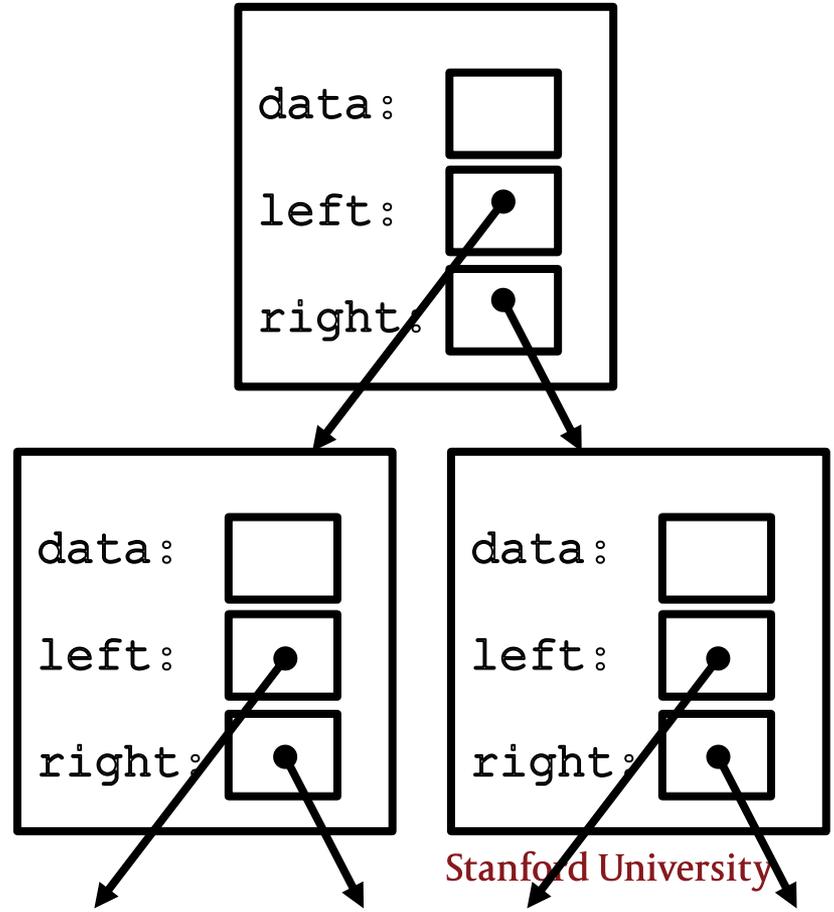
Heaps!

Binary Heaps*

Binary heaps are **one kind** of binary tree

They have a few special restrictions, in addition to the usual binary tree:

- Must be **complete**
  - › No "gaps"—nodes are filled in left-to-right on each level (row) of the tree
- Ordering of data must obey **heap property**
  - › Min-heap version: a parent's data is always ≤ both its children's data
  - › Max-heap version: a parent's data is always ≥ both its children's data

* There are other kinds of heaps as well. For example, binomial heap is extra credit on your assignment.

How many of these could be valid binary heaps?
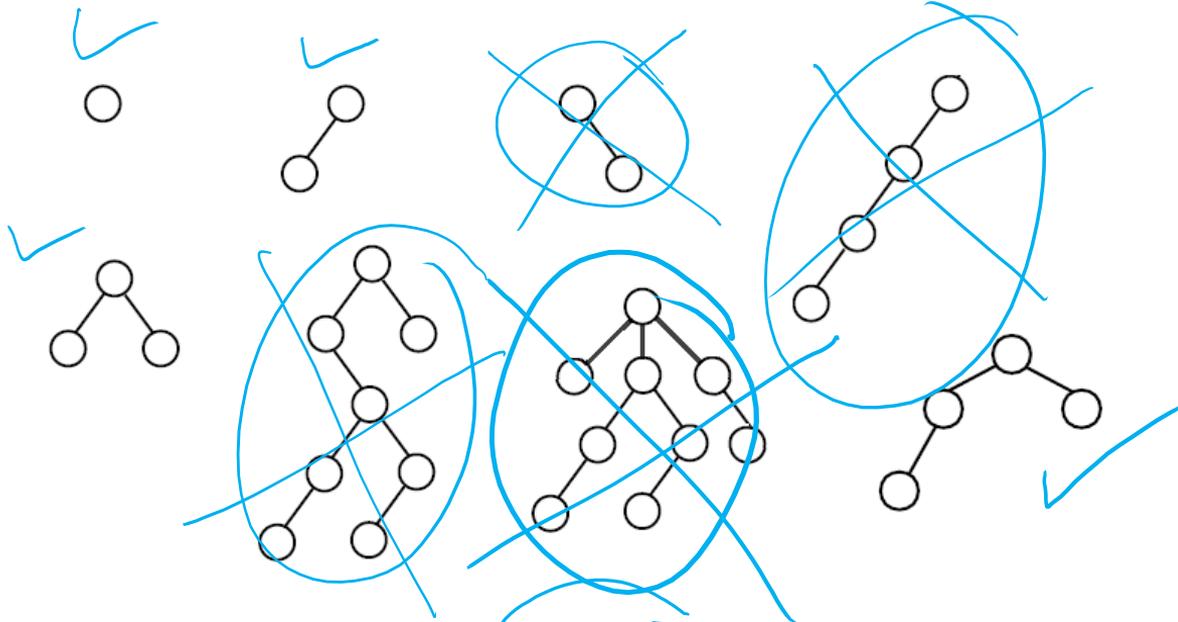
A. 0-1
B. 2
C. 3
D. 4
E. 5-8

How many of these are valid min-binary-heaps?
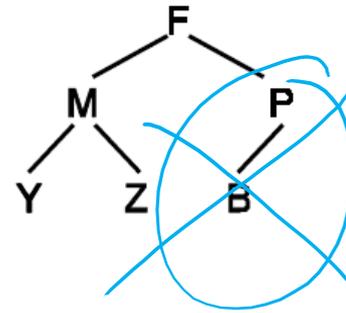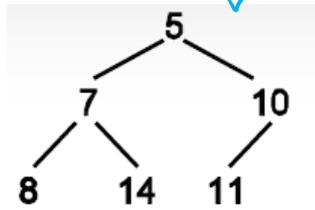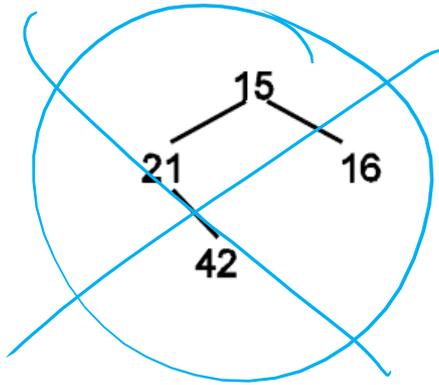
Binary heap in an array

Binary heap in an array

Binary heap is one special kind of binary tree, so we could use a node struct to represent it

However, … we actually do NOT typically use a node object to implement **heaps**

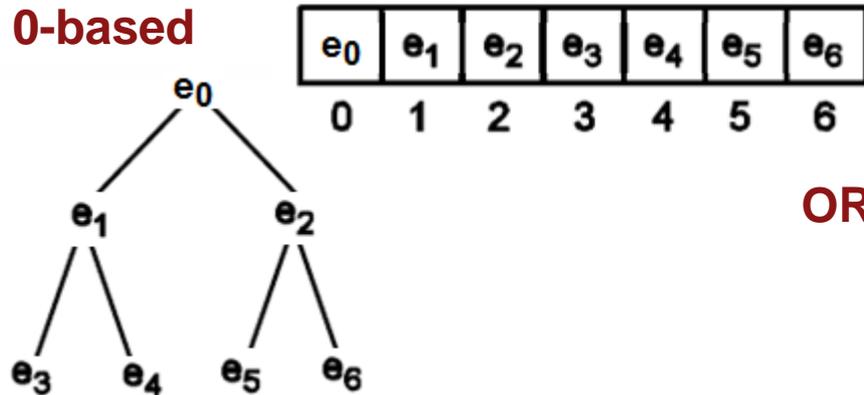Because they have the special added constraint that they must be **complete**, they fit nicely into an array
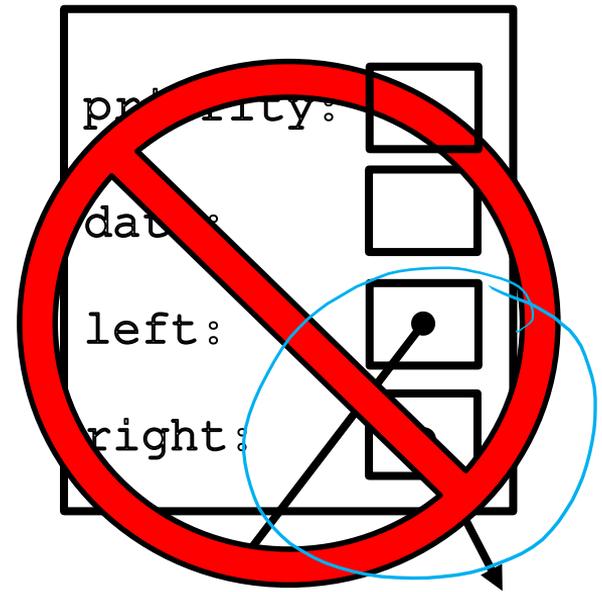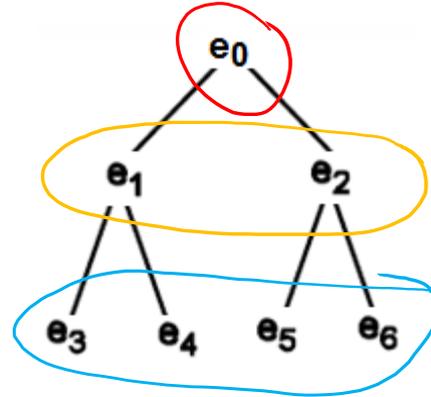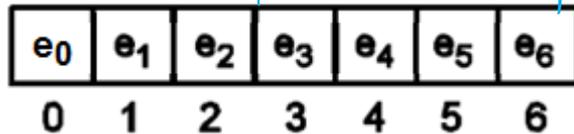
Two approaches:
Binary heap in an array

Wait, but the homework handout starts storing the elements at array index 1!

> › Either way is ok for the assignment.
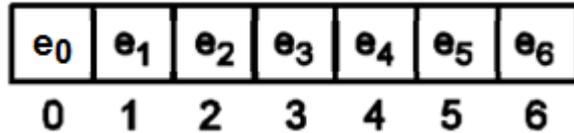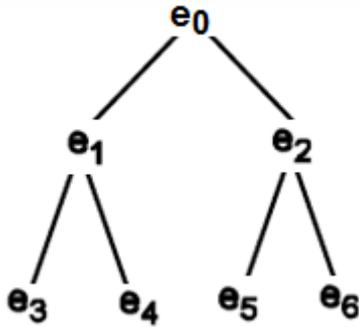> › You should understand both ways, so we're teaching both ways
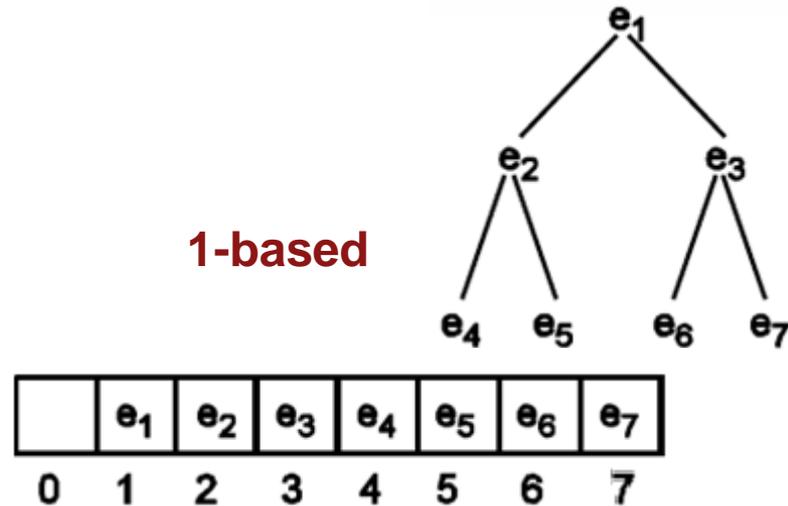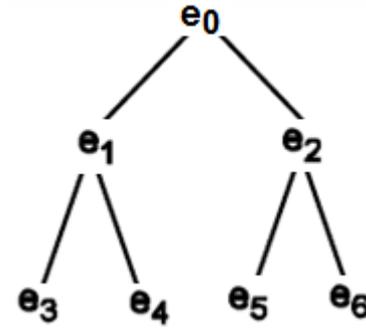
Heap in an array

For a node in array index i:

- Q: The parent of that node is found where?
- A: at index:
    - A. i – 2
    - B. i / 2
    - C. (i – 1)/2
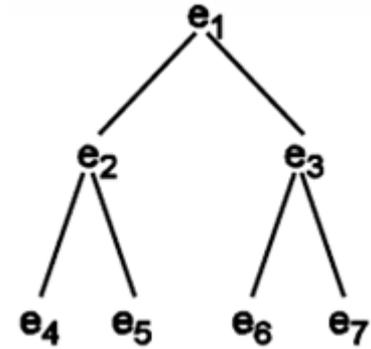   - D. 2i

Fact summary:
Binary heap in an array

0-based:

For tree of height h, array length is $2^h - 1$

For a node in array index i:

- Parent is at array index: $(i - 1)/2$
- Left child is at array index: $2i + 1$
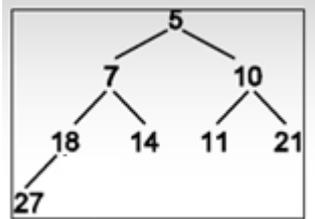- Right child is at array index: $2i + 2$

1-based:

For tree of height h, array length is 2^h

For a node in array index i:

- Parent is at array index: $i/2$
- Left child is at array index: $2i$
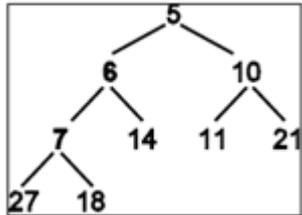- Right child is at array index: $2i + 1$

Binary heap enqueue and dequeue

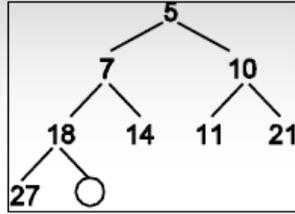# Binary heap enqueue (insert + "bubble up")

Size=8, Capacity=15

0	1	2	3	4	5	6	7	8	9	...	14
5	7	10	18	14	11	21	27	?	?	...	?

Size=9, Capacity=15

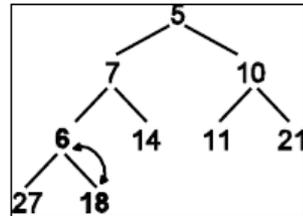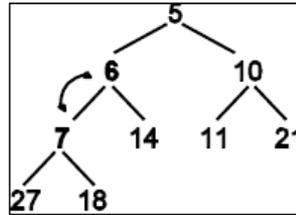| 0	1	2	3	4	5	6	7	8	9	...	14
5	6	10	7	14	11	21	27	18	?	...	?

[Binary heap insert reference page]

(a) A minheap prior to adding an element. The circle is where the new element will be put initially.
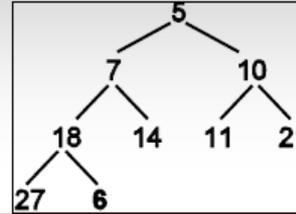
(b) Add the element, 6, as the new rightmost leaf. This maintains a complete binary tree, but may violate the minheap ordering property.
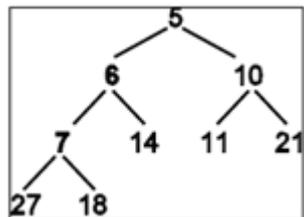
(c) "Bubble up" the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.
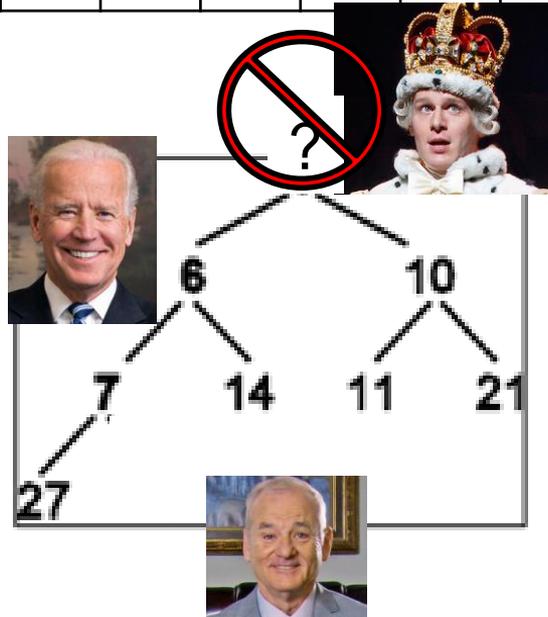
(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.

Stanford University
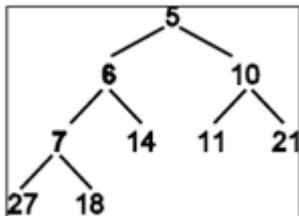
Binary heap dequeue (delete + "trickle down")

Size=9, Capacity=15

0	1	2	3	4	5	6	7	8	9	...	14
5	6	10	7	14	11	21	27	18	?	...	?

Binary heap dequeue (delete + "trickle down")

Size=9, Capacity=15

0	1	2	3	4	5	6	7	8	9	…	14
5	6	10	7	14	11	21	27	18	?	…	?
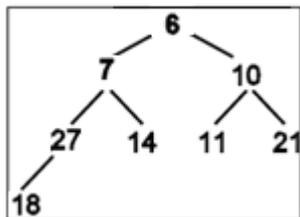
Size=8, Capacity=15

0	1	2	3	4	5	6	7	8	9	…	14
6	7	10	27	14	11	21	18	?	?	…	?

# [Binary heap delete + "trickle-down" reference page]

(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.
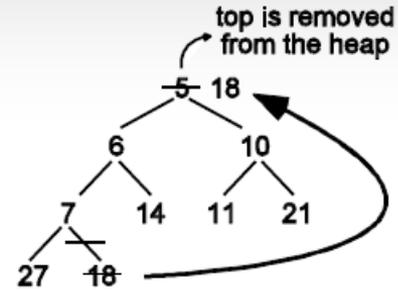
(b) "Trickle down" the element. Swapping top with the smaller of its two children leaves top's right subtree a valid heap. The subtree rooted at 18 still needs fixing.
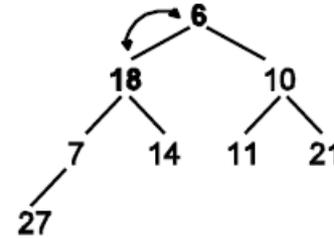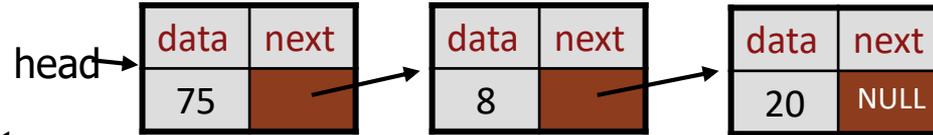
(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored

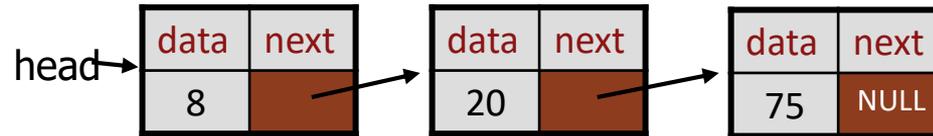ford University
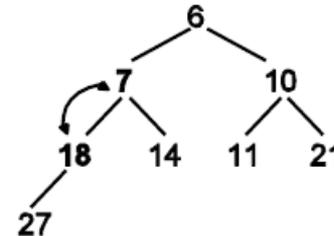
Summary analysis

Comparing our priority queue options

Some priority queue implementation options

Unsorted linked list

- Insert new element in front: O(1)
- Remove by <u>searching</u> list: O(N)

Sorted linked list

- Always insert <u>in sorted order</u>: O(N)
- Remove from front: O(1)

Priority queue implementations

We want the best of both

Fast add AND fast remove/peek

We will investigate trees as a way to get the best of both worlds
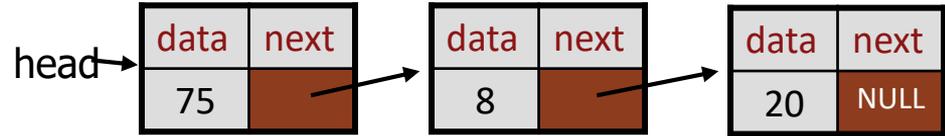

Fast add   +   Fast remove/peek   =   🙂
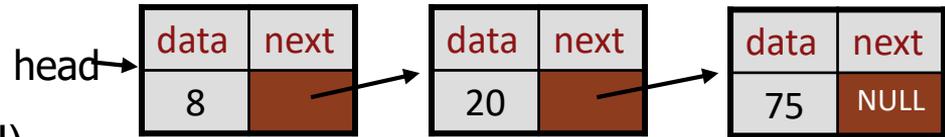
Review: priority queue implementation options

Unsorted linked list

- Insert new element in front: O(1)
- Remove by <u>searching</u> list: O(N)

head → | data | next |
       | 75 | → | data | next |
              | 8 | → | data | next |
                     | 20 | NULL |

Sorted linked list

- Always insert <u>in sorted order</u>: O(N)
- Remove from front: O(1)

head → | data | next |
       | 8 | → | data | next |
             | 20 | → | data | next |
                     | 75 | NULL |

Binary heap

- Insert + "bubble up": O(   )
- Delete + "trickle down": O(   )