

Midterm 1 Review



CSCI 104

Runtime

Recall our formulas:

- Arithmetic Series:
$$\sum_{i=0}^n \Theta(i^p) = \Theta(n^{p+1})$$

- Geometric Series:
$$\sum_{i=0}^n c^i = \Theta(c^n)$$

- Harmonic Series:
$$\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$$

Runtime

Let's determine the runtime of this algorithm:

Algorithm 1 Number Filtering

```
1: Input: a positive integer  $n \geq 2$ 
2: initialize the Boolean array nums such that  $nums(i) = \mathbf{True}$  for  $i = 2, \dots, n$ 
3: for  $i = 2, \dots, n$  do
4:   for  $j = 2, \dots, \lfloor \frac{n}{i} \rfloor$  do
5:     if  $i * j \leq n$  then
6:        $nums(i * j) = \mathbf{False}$ 
7:     end if
8:   end for
9: end for
```

Runtime

Let's walk through it line-by-line!

Algorithm 1 Number Filtering

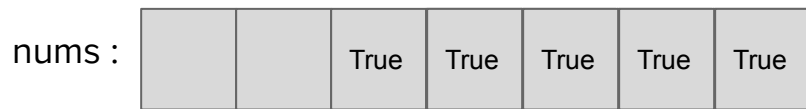
```
1: Input: a positive integer  $n \geq 2$ 
2: initialize the Boolean array nums such that  $nums(i) = \mathbf{True}$  for  $i = 2, \dots, n$ 
3: for  $i = 2, \dots, n$  do
4:   for  $j = 2, \dots, \lfloor \frac{n}{i} \rfloor$  do
5:     if  $i * j \leq n$  then
6:        $nums(i * j) = \mathbf{False}$ 
7:     end if
8:   end for
9: end for
```

- Line 2: We declare an array of size n : how much time does this take?

nums :

		True	True	True	True	True
--	--	------	------	------	------	------

Runtime



Algorithm 1 Number Filtering

```
1: Input: a positive integer  $n \geq 2$ 
2: initialize the Boolean array  $nums$  such that  $nums(i) = \mathbf{True}$  for  $i = 2, \dots, n$ 
3: for  $i = 2, \dots, n$  do
4:   for  $j = 2, \dots, \lfloor \frac{n}{i} \rfloor$  do
5:     if  $i * j \leq n$  then
6:        $nums(i * j) = \mathbf{False}$ 
7:     end if
8:   end for
9: end for
```

- Outer for loop will go from $i: 2 \rightarrow n$:
 - ($i = 2$): $j: 2 \rightarrow n/2$
 - ($i = 3$): $j: 2 \rightarrow n/3$
 - ($i = 4$): $j: 2 \rightarrow n/4$
 - What is the pattern here?
- What is the work inside the nested for loop?
- What is the total runtime of the nested for loops?

BONUS: What does this algorithm do?

Recursion

Given the heads of two sorted linked lists, return the merged, sorted linked list.

Adapted from merge sort.

```
1 struct Node {  
2     Node* next;  
3     int val;  
4 };  
5  
6 class Merge {  
7     public:  
8         Node* merge(Node* l1, Node* l2);  
9     private:  
10        Node* head;  
11 };
```

```
4 Node* Merge::merge(Node* l1, Node* l2) {  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```



Recursion

Step 1: What is the base case? *Hint: there may be more than 1 base case*

Base Cases: If one of the lists is null.

What should happen? Return the non-null list. (or just return null if both are null)

Recursion

Step 2: What is the recursive case?

Recursive Case: We need to update the next pointer without losing track of the head of the list we will return.

What should happen? How do we do that?

1. First, check which list currently has the smaller value at the front
2. For whichever list has the smaller head, then we want to:
 - a. Update that node's NEXT (everything behind it) recursively – this is the trust fall!
 - b. Return that pointer

Recursion

```
Node* Merge::merge(Node* l1, Node* l2) {
    // BASE CASE: l1 is empty, so return l2
    if(!l1) {
        return l2;
    }

    // BASE CASE: l2 is empty, so return l1
    if(!l2) {
        return l1;
    }

    // IF l1 head is lower, then
    // 1. update l1's list "behind" the head
    // 2. return the head
    // do the same but flipped if l2 head is lower
    if(l1->val < l2->val) {
        l1->next = merge(l1->next, l2);
        return l1;
    } else {
        l2->next = merge(l1, l2->next);
        return l2;
    }
}
```

Head vs. Tail Recursion

- Tail recursion is when you **return the function call**, i.e. there is no more code to execute below the recursive call
 - Ex. `return recursiveFunc(int param)`
- Tail recursion is great because then the program doesn't have to "go back" to that call to execute code
 - Functions are basically a collection of statements that sit on the call stack
 - By using tail recursion instead of head, there's only one version of the function sitting on the call stack
- Whenever there is code below a return statement to be executed, i.e. after making the recursive call the program has to revisit that instance of the call, then it's head recursion
 - There can multiple versions of the function sitting on the call stack, so is less efficient both time and memory wise

ADTs

Type	Key Points	Operation Runtimes	Examples
List	Ordered, access based on position (index), may contain duplicates	Depends on implementation	A music album, a book series
Stack	LIFO (last in first out)	Push, pop, top All $O(1)$ if implemented right	A stack of trays at a cafeteria, tandem parking lot, "undo"
Queue	FIFO (first in first out)	Push back, pop front, front All $O(1)$ if implemented right	Waiting in line for a cashier at a store, printing a document
Map	Stores key, value pairs, keys must be unique (but values can have duplicates), no inherent ordering	Insert, remove, find/lookup: All $O(\log n)$	A dictionary, a grade book (key: student, value: grade)
Set	Like a map but only store keys, items must be unique, accessed based on keys, not indices	Insert, remove, find/lookup: All $O(\log n)$	Spices in a cupboard, all USC courses
Priority Queue	Self-orders items based on value, only allows retrieval/removal of the "best" priority item	Push, pop, top $O(\log n)$, $O(\log n)$, $O(1)$	A hospital where patients arrive with injuries of varying severity

ADTs

1. In operating systems, multiple processes can happen concurrently with the use of threads. What ADT could you use to store processes, where they are assigned to threads based on the order in which they were first requested?
 - a. What if processes were now instead handled by their level of importance?
2. Say you're a teacher and you want to keep track of your students and their emergency contacts. What data structure should you use?
3. Say you want to keep track of every single line of code you write in a C++ class. What data structure should you use?
4. Say you're trying to write a simple Towers of Hanoi implementation. What data structure should you use for each peg?

**In Towers of Hanoi, you can only remove the top ring*

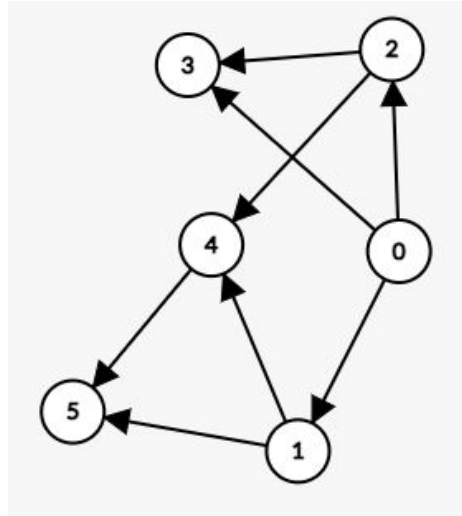


ADTs

1. Queue of processes. Ex. `std::queue<Process*>`
 - a. Priority queue of processes. `std::priority_queue<Process*>`. Heap could also work.
2. Map, where students are the key and the emergency contact is the value. Ex. `std::map<Student*, EmergencyContact*>`
3. List of all the lines, since C++ is executed sequentially. Use the index as the line number. Ex. `std::list<std::string>`
4. The pegs are handled last in, first out, so it should be a stack of rings. Ex. `std::stack<Ring*>`

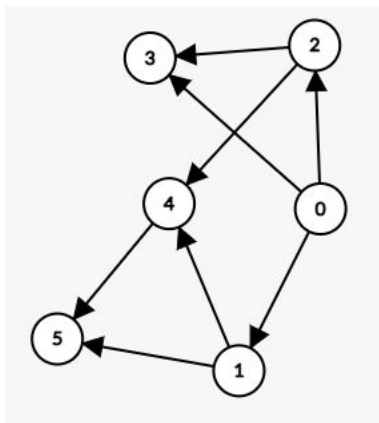
Stacks & Queues

Given this directed graph, use a stack and a queue to find a path from node 0 to node 3. Show the current state of the stack and queue in each iteration of the search and break ties from the node's value, pushing higher numbered neighbors on first. Do not revisit nodes!



Stack

Given this directed graph, use a stack and a queue to find a path from node 0 to node 3. Show the current state of the stack and queue in each iteration of the search and break ties from the node's value (higher #s first). Do not revisit nodes!

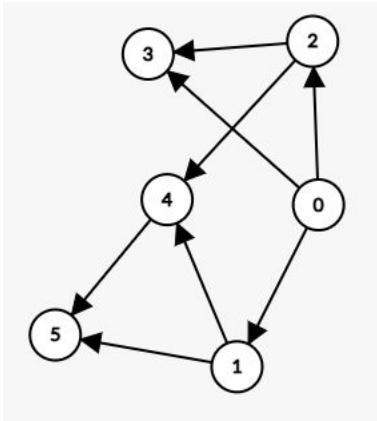


Initial Stack:	Next:	Next:	Next:	Next:	Next:
0	1	4	5	2	3
-----	2	5	2	3	3
	3	2	3	-----	-----
	-----	3	-----		
Pop 0, push neighbors	Pop 1, push neighbors	Pop 4, push neighbors (none)	Pop 5, push neighbors (none)	Pop 2, push neighbors	Pop 3; found!

Note- depending on when you check if/mark a node visited, you may/may not push 3 on in the second to last step!

Queue

Given this directed graph, use a stack and a queue to find a path from node 0 to node 3. Show the current state of the stack and queue in each iteration of the search and break ties from the node's value (higher #s first). Do not revisit nodes!



Initial Queue:

0

Pop 0, push
neighbors

Next:

| 1 |
| 2 |
3

Pop 3; found!

Heaps

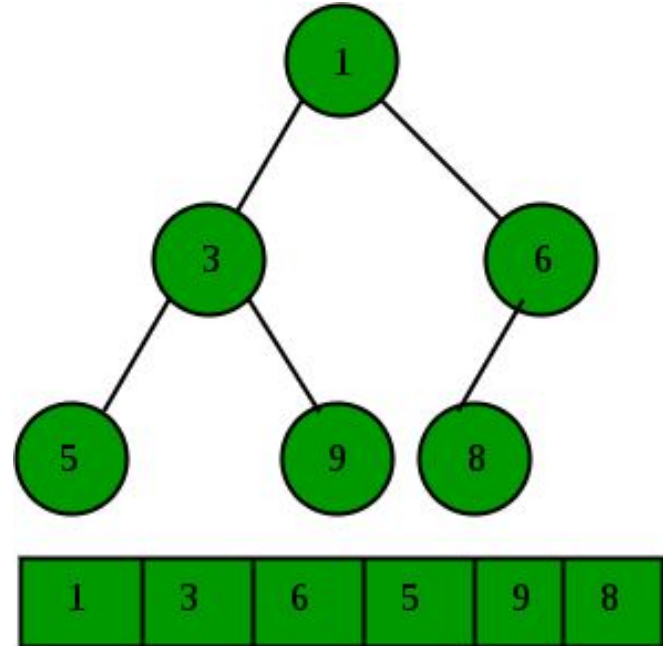
Review: Store a heap in an array

Array starting at index 0, given location i :

Parent Location: $(i - 1) / 2$

Left Child: $2i + 1$

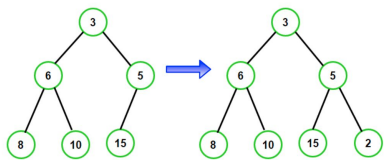
Right Child: $2i + 2$



Pushing + Popping a Heap

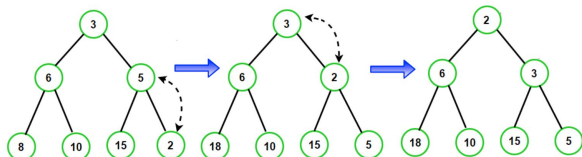
Push:

1. Insert to next index (bottom of tree)
2. Swap with parent until it's not "better than" its parent, or is now the root



Push(2) called on min heap

Add the new element 2 to the bottom level of the heap and call Heapify-up(2)



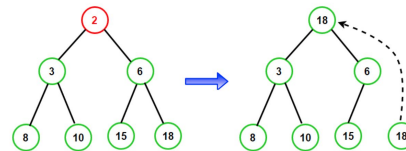
Swap node 2 with its parent as heap property is violated
swap(5, 2)

Swap node 2 with its parent as heap property is still violated
swap(3, 2)

Resultant Min Heap

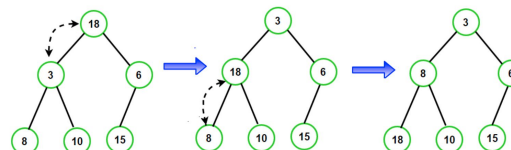
Pop:

1. Swap 0th element (thing to be popped) with last element, delete
2. Swap root element down til in correct spot



Pop() called on min heap

Replace the root of the heap with the last element on the last level and call Heapify-down(root)



Swap root node with its smaller child
swap(18, min(3, 6))

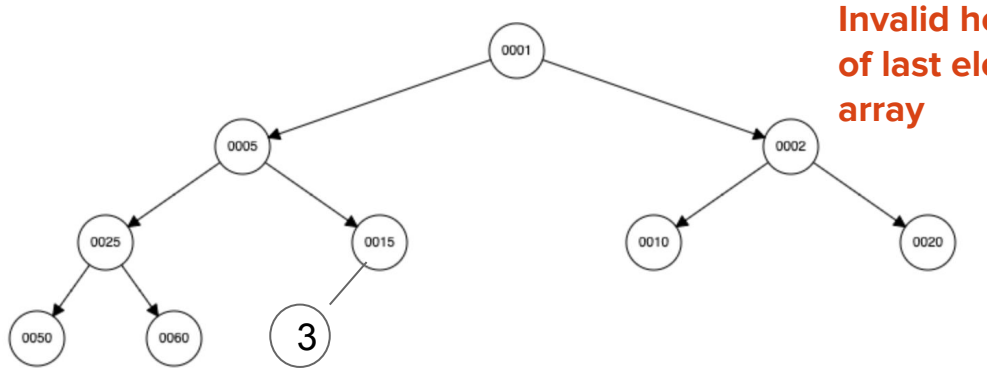
Swap node 18 with its smaller child
swap(18, min(8, 10))

Resultant Min Heap

Heap Example

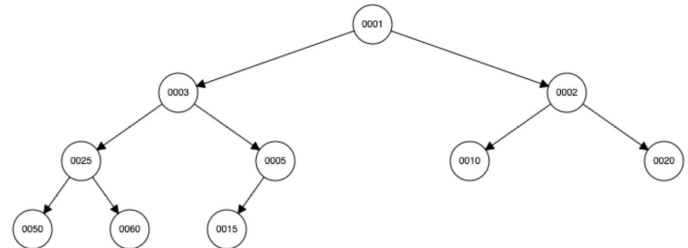
Given the array: [1, 5, 2, 25, 15, 10, 20, 50, 60, 3]

What does it look like as a binary heap in tree form? Is it a valid heap? If it is, what type of heap? If it isn't, what swap(s) do we need to make to make it valid?



Invalid heap because of last element in the array

Make it a min heap by swapping 3 up twice



Inheritance

Explain the difference between public, protected, and private inheritance.

How does the visibility of inheritance affect:

- Public member variables?
- Protected member variables?
- Private member variables?

Inheritance Example

- What can Island access?
- What can Atoll access?

Island:

- **GetArea(), GetFeatureName(), setArea(), setFeatureName()**

Atoll:

- **GetArea(), GetFeatureName(), setArea(), setFeatureName(), GetOcean()**

```
class GeographicFeature {
public:
    virtual int GetArea() = 0;
    std::string GetFeatureName();
protected:
    void setArea(int h);
    void setFeatureName(std::string fn);
private:
    int area;
    std::string featureName;
}

class Island : public GeographicFeature {
public:
    std::string GetOcean();
private:
    std::string surroundingOcean;
}

class Atoll : public Island {
public:
    std::vector<std::string> GetCoralTypes();
private:
    std::vector<std::string> coralTypes;
}
```

Inheritance Example

Island is now a PRIVATE GeographicFeature.

- What can Island access?
- What can Atoll access?

Island:

- **GetArea(), GetFeatureName(), setArea(), setFeatureName()**

Now all private

Atoll:

- **GetOcean()**

```
class GeographicFeature {
public:
    virtual int GetArea() = 0;
    std::string GetFeatureName();
protected:
    void setHeight(int h);
    void setFeatureName(std::string fn);
private:
    int area;
    std::string featureName;
}

class Island : private GeographicFeature {
public:
    std::string GetOcean();
private:
    std::string surroundingOcean;
}

class Atoll : public Island {
public:
    std::vector<std::string> GetCoralTypes();
private:
    std::vector<std::string> coralTypes;
}
```

Iterators

- Way to iterate (i.e. visit) every single item in a collection
- Usually, sets and maps are **unordered** collections of data, i.e. there is no obvious way to visit the data, unlike a vector (array)
- Iterators are objects that point, usually temporarily, at an object within a container; good way to abstract the method that is being used to iterate, so you don't personally have to code that!
- Ex.
 - `std::set<int>::iterator it;`
`for(it = mySet.begin(); it != mySet.end(); ++it)`
`Std::cout << *it << std::endl;`

Tips + Questions

- If you get stuck on a problem, **move on!** You may know how to do other ones, so try to get as many points as possible, then come back if you have time
- Topics especially to note: ADTs, basic recursion, inheritance, runtime analysis
- Recommended resources: lecture notes, PAs (generally), Practice Midterm!!!!
- Questions?