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 $nums(i)$ = **True** for $i = 2, ..., n$
3: **for** $i = 2, \ldots, n$ **do**
4:     **for** $j = 2, \ldots, \lfloor \frac{n}{i} \rfloor$ **do**
5:         **if** $i * j \leq n$ **then**
6:             $nums(i * j)$ = **False**
7:         **end if**
8:     **end for**
9: **end for**

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  }
```

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

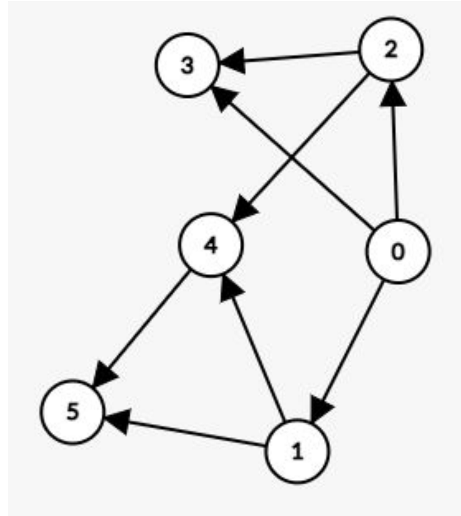
ADTs

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*

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!



Heaps

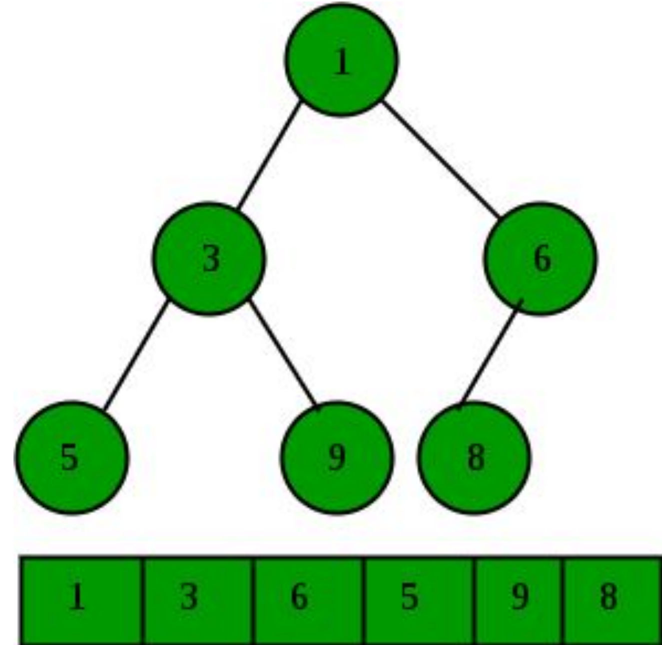
Review: Store a heap in an array

Array starting at index 0, given location i:

Parent Location: (i - 1) / 2

Left Child: 2i + 1

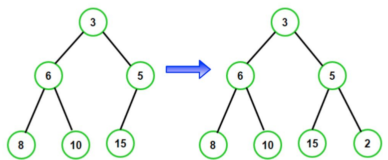
Right Child: 2i + 2

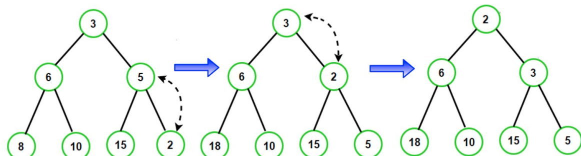


Pushing + Popping a Heap

Push:

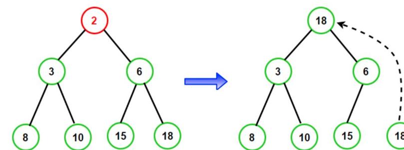
1. Insert to next index (bottom of tree)
2. Swap with parent until it's not "better than" its parent, or is now the root

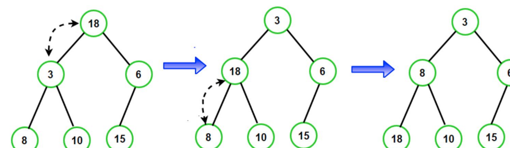




Pop:

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





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?

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?

```
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?

```
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?