UPE Tutoring:

CS 32 Final Review

Sign-in http://bit.ly/2xeQMKx

Slides link available upon sign-in

Table of Contents

Sorting: `O(n^2)` algorithms

Sort Name	Stability	Notes
Selection Sort	Unstable	Always `O(n^2)`, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow).
Insertion Sort	Stable	`O(n)` for already or nearly-ordered arrays. `O(n^2)` otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	`O(n)` for already or nearly-ordered arrays (with a good implementation). `O(n^2)` otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	Approximately `O(n^1.25)`. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.

Sorting: `O(nlogn)` algorithms

Sort Name	Stability	Notes
Quicksort	Unstable	`O(nlogn)` average, `O(n^2)` for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up `O(n)` space (for recursion) in the worst case, `O(logn)` space average.
Mergesort	Stable	`O(nlogn)` always. Used for sorting large amounts of data on disk (aka “external sorting”). Can be used to sort linked lists. Can be parallelized across multiple cores. However, requires `O(n)` space for merging – other sorts don’t need extra RAM.
Heapsort	Unstable	`O(nlogn)` always. Sometimes used in low-RAM embedded systems because of its performance/low memory requirements.

Sorting: Quick tips

- Print out Nachenberg's sorting cheat sheet!
- Assuming that we are sorting numbers in increasing order:
  - One pass of Bubble Sort will move the largest item to the end.
  - One pass of Selection Sort will move the smallest item to the start.
  - After n passes of Insertion Sort, the first n items will be in sorted order (as if we completely sorted an array of size n).
  - Bubble Sort, Insertion Sort, and Selection Sort are good for simplicity.
  - Mergesort and Quicksort are good for efficiency.
  - Heapsort and Shellsort are unlikely to be on the exam: understand them and definitely bring notes.

Examples

The following example problems will have this format:

```
    [  Original Array  ]
    [ After Two Passes ]
```

Q: Which of the following algorithms may have been used?

Assume we are sorting in increasing order.

A. Selection Sort
B. Bubble Sort
C. Insertion Sort
D. More than one of the above

Examples

Q1	**45**	**40**	**42**	**47**	**43**	**41**
	40	41	42	47	43	45

Q2	**45**	**40**	**42**	**47**	**43**	**41**
	40	45	42	47	43	41

Q3	**45**	**40**	**42**	**47**	**43**	**41**
	40	42	43	41	45	47

Solutions

1. A – Selection sort
    a. After 2 passes, first 2 elements of final array are in sorted order
2. C – Insertion sort
    a. After 2 passes, first 2 elements from original array are in sorted order
3. B – Bubble sort
    a. After 2 passes, the last 2 elements of the final array are in sorted order

Big-O

- Take the highest order, drop the less significant terms
- Drop constants
- Keep answer in terms of input variables
    - Be wary of the *name* and *number* of inputs you're dealing with
- Space complexity is also a thing (but you will mainly be dealing with time complexity)

```
Running time:
1,000n^3 + 10,000,000,000n + sqrt(m)
```

```
O(n^3 + sqrt(m))
```

Big-O Example

```
int add100(vector<int> array) {
     if (array.size() < 100)
          return 0;

     int sum = 0;

     for (int i = 0; i < 100; i++)
          sum += array[i];

     return sum;
}
```

Answer Bank

O(1)
O(logn)
O(n)
O(nlogn)
O(logm)
O(m)
O(mlogm)

Big-O Example

```
int add100(vector<int> array) {
     if (array.size() < 100)
          return 0;

     int sum = 0;

     for (int i = 0; i < 100; i++)
          sum += array[i];

     return sum;
}
```

Answer Bank

O(1)
O(logn)
O(n)
O(nlogn)
O(logm)
O(m)
O(mlogm)

Answer: O(1)

More Big-O Examples

1. Add `n` numbers together
2. Find min of `n` sorted numbers
3. Sort `n` random integers
4. Find item in STL `set` of `m` integers
5. Insert an element to beginning of STL `list` of `m` integers
6. Sort `n` ages of people

Answer Bank

```
O(1)
O(logn)
O(n)
O(nlogn)
O(logm)
O(m)
O(mlogm)
```

More Big-O Examples

1. Add n numbers together
2. Find min of n sorted numbers
3. Sort n random integers
4. Find item in STL `set` of `m` integers
5. Insert an element to beginning of STL `list` of `m` integers
6. Sort n ages of people

Answer Bank

```
O(1)
O(logn)
O(n)
O(nlogn)
O(logm)
O(m)
O(mlogm)
```

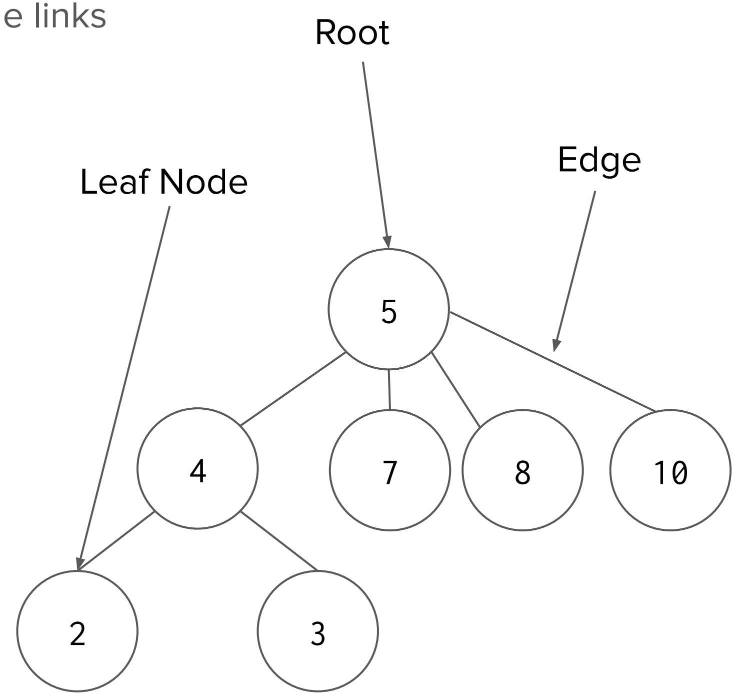
```
Answers:
O(n), O(1), O(nlogn), O(logm), O(1), O(n)
```

Data Structures: trees

- A tree is the idea of a node linked to multiple children. The links to each child from a parent node is called a branch.
- A tree with only one branch is essentially a Linked List.
- A typical node can be defined like so:

```
template <typename T>;
struct Node {
     Node(T value): val(value)
     T val;
     vector<Node*> children;
};
```

Note: Trees are *acyclic*, meaning there is no cycle of nodes.

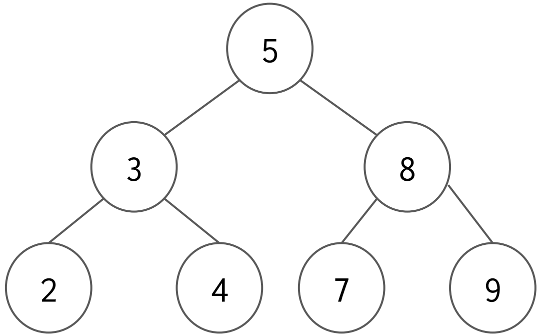


Binary Search Trees

Binary Search Trees are probably the most common data structures you will deal with.

- Each node only has two children.
- The left node’s value is smaller and the right node’s value is greater.
- They are based off of the idea of using binary search, so that we can eliminate half of the tree once we know what we’re looking for.
- They are ordered.

```
struct BSTNode {
  BSTNode(int value):
    val(value), right(nullptr),
    left(nullptr) {}
  int val;
  Node* right;
  Node* left;
};
```



Search

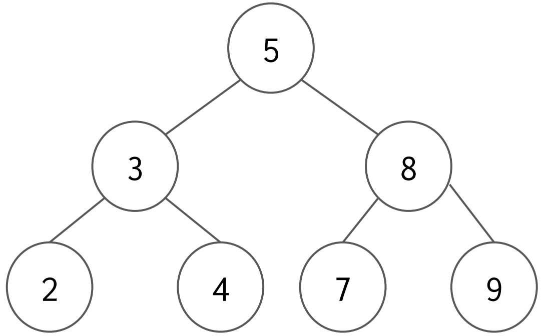
- Start at the root of the tree, and continue until we hit NULL:
  - If V is the same as the node's value, then we have found the node!
  - If V is greater than the node's value, go right.
  - If V is less than the node's value, go left.
  - If we hit a NULL then we didn't find our value, V.
- Searching in a BST of size `n` generally has a time complexity of `O(logn)`
  - This is because each movement usually halves the data set we have to consider
  - Example: `|data set| = n`, then `n/2`, then `n/4`, then ..., then `1`. This takes `log_2(n)` steps until we only consider 1 element.

Tips and Tricks to look out for with BST

There are four main ways of traversing a BST:

- Pre-Order Traversal (CLR) - Look at current node, then left then right (DFS).
- In-Order Traversal (LCR) - Look at left node, then current, then right (DFS).
- Post-Order Traversal (LRC) - Look at the left node, then right, then current (DFS).
- Level-Order Traversal - Look at levels of nodes making your way down the entire tree - use a queue to store the left and right nodes and then analyze those (BFS).

**DFS - Depth First Search: Go down a branch all the way and then make your way back*
**BFS - Breadth First Search: Look at node and then left and right nodes, level by level.*



Pre-Order output:	5 3 2 4 8 7 9
In-Order output:	2 3 4 5 7 8 9
Post-Order output:	2 4 3 7 9 8 5
Level-Order output:	5 3 8 2 4 7 9

More Tips and Tricks for BST

Most solutions for BST based algorithms are best solved using the ~~dreaded~~ wonderful recursion! Your base case will probably `if(root == nullptr)`, to make sure that you've reached the end of a branch.

Another tip is to use helper functions with recursion if you need fewer or more parameters to make your recursive algorithm work.

Lets do some BST questions!

Practice Question: Find the maximum depth of a BST

Given a binary search tree, return the maximum depth of the tree as an integer.

Solution: Find the maximum depth of a BST

```
int maxDepth(Node* root) {
  if (root == nullptr) { return 0; }       // If no node, then depth is zero.
  int leftSum = 1 + maxDepth(root->left);  // Add 1 to account for the root node.
  int rightSum = 1 + maxDepth(root->right);
  return (leftSum >= rightSum) ? leftSum : rightSum;
}
```

Practice Question: Return if a tree is balanced

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1.

Solution: Return if a tree is balanced

```
#include<cmath>
bool isBalanced(Node* root) {
  if(root == nullptr) { return true;}
  int L = maxDepth(root->left);
  int R = maxDepth(root->right);
  int diff = abs(L - R);
  return diff <= 1 &&
      isBalanced(root->left) &&
      isBalanced(root->right);
}
```

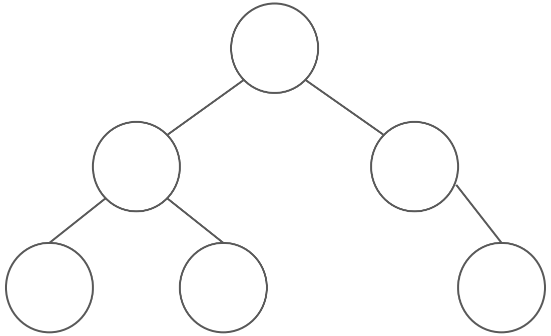
```
int maxDepth(Node* root) {
  if(root == nullptr) { return 0; }
  int leftSum =
      1 + maxDepth(root->left);
  int rightSum =
      1 + maxDepth(root->right);
  return (leftSum >= rightSum) ?
      leftSum :
      rightSum;
}
```

Practice Question: IALNSQ

Really pertinent to the CS32 final.

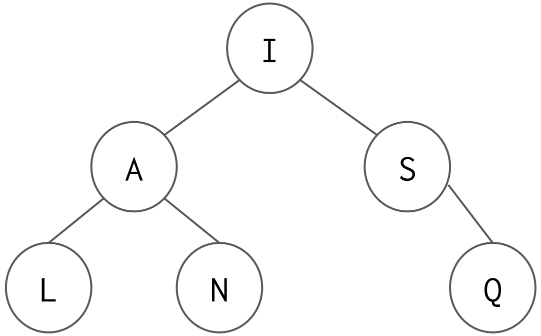
Reconstruct the given binary tree if the string "`IALNSQ`" was given as an output for:

A. Pre-Order Traversal
B. In-Order Traversal
C. Post-Order Traversal

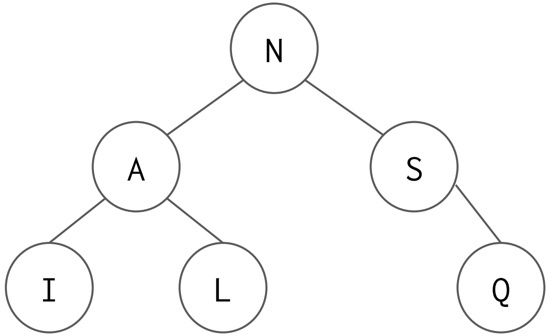


Solution: IALNSQ

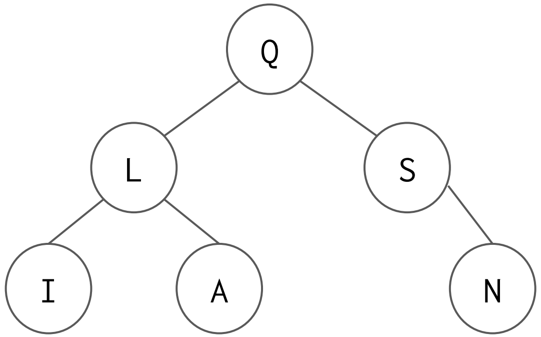
Pre-Order (CLR):



In-Order (LCR):

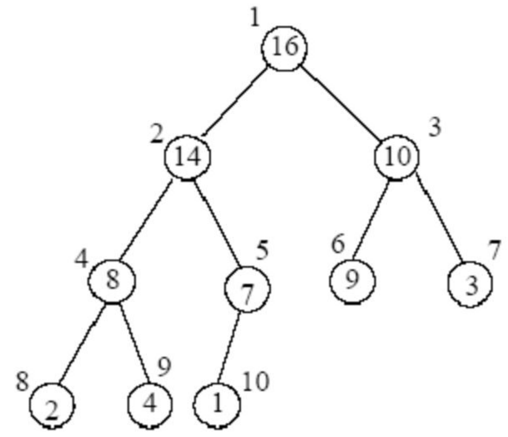


Post-Order (LRC):



Heaps

- A heap is just a modified tree data structure that satisfies either the max or min heap property
- Max heap: the parent node's value is greater than that of its children (shown on the right)
- Min heap: the parent node's values is less than that of its children
- A heap can be visually represented as a tree!



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Lets consider the *i*:th **node** in a Heap that has the value A[i]

`PARENT(i) = i/2`	Return the index of the father node
`LEFT(i) = 2i`	Return the index of the left child
`RIGHT(i) = 2i+1`	Return the index of the right child

Hash tables: a motivating example

Consider the fairly common interview problem:

Let's start with something simple.

Hash tables: a motivating example

Consider the fairly common interview problem:

Let's start with something simple.
Write a function that takes a string and returns true if any two characters are equal.

**Let's assume all characters are lowercase.*

Hash tables: a motivating example

```
bool containsDuplicate(string s) {
  for (int i = 0; i < s.length; i++)
    for (int j = i + 1; j < s.length; j++)
      if (s[i] == s[j])
        return true;

  return false;
}
```

Hash tables: a motivating example

```
bool containsDuplicate(string s) {                   // O(n^2)
  for (int i = 0; i < s.length; i++)                 // n elements
    for (int j = i + 1; j < s.length; j++)              // n elements
      if (s[i] == s[j])
        return true;

  return false;
}
```

Hash tables: a motivating example

```
bool containsDuplicate(string s) {
  int count[26] = { 0 };
  for (int i = 0; i < s.length; i++)
    count[s[i] - 'a']++;
  for (int i = 0; i < 26; i++)
    if (count[i] > 1) return true;

  return false;
}
```

Hash tables: a motivating example

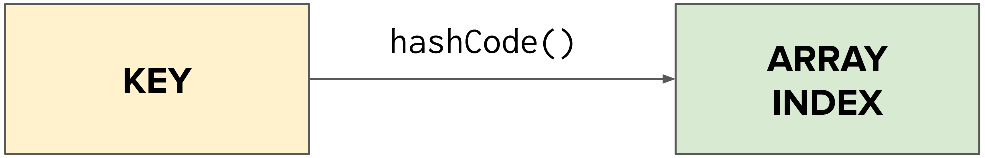
```
bool containsDuplicate(string s) {   // O(n) algorithm!
  int count[26] = { 0 };
  for (int i = 0; i < s.length; i++) // n elements
    count[s[i] - 'a']++;
  for (int i = 0; i < 26; i++)       // k elements
    if (count[i] > 1) return true;

  return false;
}
```

Hash tables: a motivating example

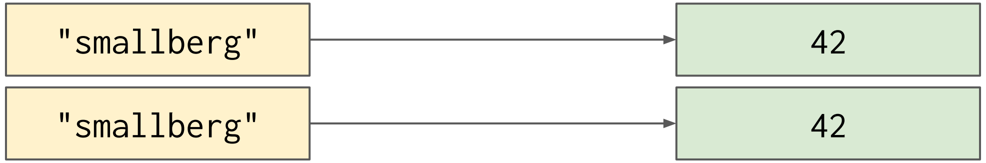
- In principle, we use the `count` array as a hash table.
    - The `count` array maps a given character to an integer.
    - Our hash function is the ASCII encoding for each character.
    - We take advantage of an array's `O(1)` access time complexity.
- At the heart of it, we use hash tables because we want to take advantage of the trusty array's incredibly fast access time.

Hash functions



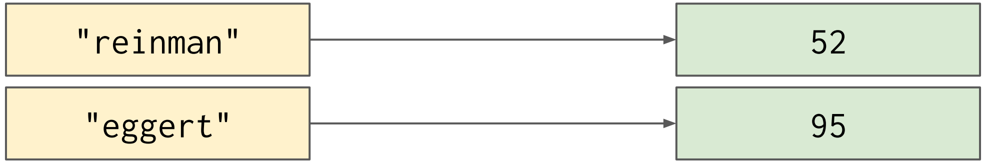
- Hash functions provide key-to-index mappings.
- There are two properties that need to be satisfied by a hash function:

Hash functions



- Hash functions provide key-to-index mappings.
- There are two properties that need to be satisfied by a hash function:
    - Consistency. For a given input, you should always get the same index.

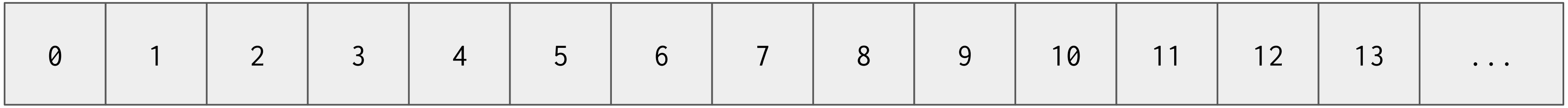
Hash functions



**May or may not reflect their relative difficulties.*

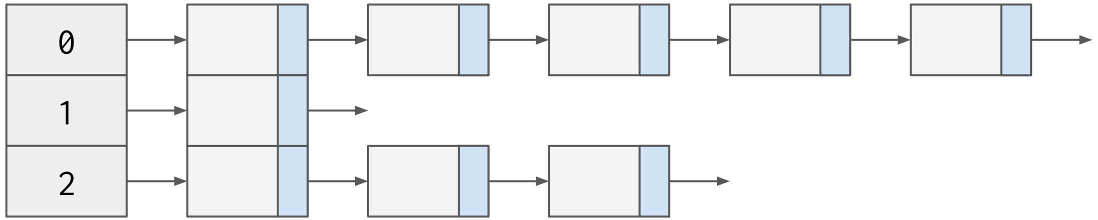
- Hash functions provide key-to-index mappings.
- There are two properties that need to be satisfied by a hash function:
  - Consistency. For a given input, you should always get the same index.
  - Well-distributed. The hash function should return different indices for any two keys.

Collisions



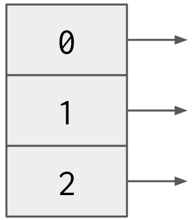
- In an ideal world, we would have infinitely long arrays. Then for things like integers, we could just use the identity function to hash them.

Collisions



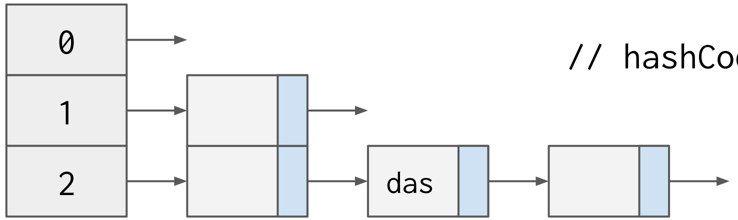
- In an ideal world, we would have infinitely long arrays. Then for things like integers, we could just use the identity function to hash them.
- Unfortunately, we don't. And even if we did, our hash functions would probably give us colliding integers anyways. Instead, we generally use linked lists to do something called "collision chaining."

Collisions



- The algorithm for hash tables can be summarized as follows:
  - To initialize, make an array of linked lists.

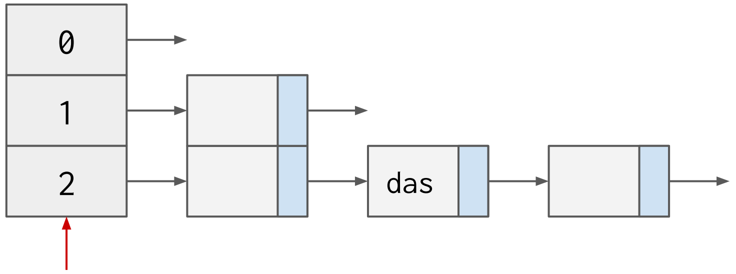
Collisions



```
// hashCode("das") returns 2.
```

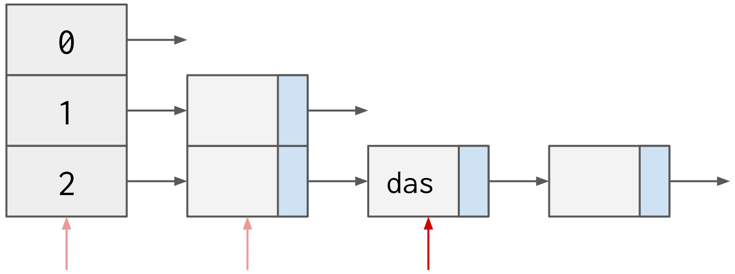
- The algorithm for hash tables can be summarized as follows:
    - To initialize, make an array of linked lists.
    - To lookup or insert into a linked list:
        - Use the hash function to find the corresponding bucket index

Collisions



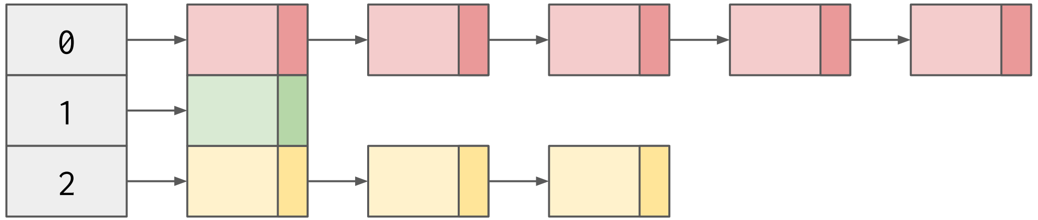
- The algorithm for hash tables can be summarized as follows:
  - To initialize, make an array of linked lists.
  - To lookup or insert into a linked list:
    - Use the hash function to find the corresponding bucket index
    - Jump directly to the bucket head

Collisions



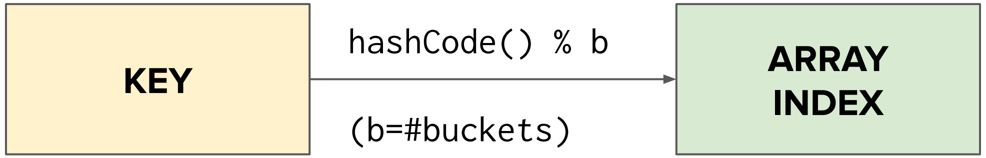
- The algorithm for hash tables can be summarized as follows:
    - To initialize, make an array of linked lists.
    - To lookup or insert into a linked list:
        - Use the hash function to find the corresponding bucket index
        - Jump directly to the bucket head
        - Linearly traverse over the linked list

Collisions



- For this reason, we want our hash functions to map to different buckets. If we have too many collisions – or in the pathological case – our hash table will have a lookup time complexity of `O(n)`. Not great.
- Conversely, with a perfect hash function, we achieve a time complexity of `O(1 + n/k)`, which generally reduces to `O(1)`.

Writing hash functions



- Previously, we gave a slightly simplified model for hash functions!
- On top of providing a corresponding integer for a given key, we need to apply the modulo operator to find an actual bucket.

Writing hash functions

```
a[n] * pow(31, n) + a[n - 1] * pow(31, n - 1) + ... + a[0]
```

- Previously, we gave a slightly simplified model for hash functions!
- On top of providing a corresponding integer for a given key, we need to apply the modulo operator to find an actual bucket.
- When writing a hash function for strings, consider using the position as well as the character value itself when computing a corresponding integer.

Comparison with binary search trees

	Binary Search Tree ***Ordered!***	**Hash Table** ***Unordered!***
Access	`O(log(n))`	`O(1)`
Search	`O(log(n))`	`O(1)`
Insertion	`O(log(n))`	`O(1)`
Deletion	`O(log(n))`	`O(1)`

Priority Queue

- A priority queue is an abstract data type like a queue or stack, but with priority associated with its element
- An element with high priority is served before an element with low priority
- A priority queue can be implemented with heaps
- Provides constant lookup of highest-priority object, with expense of logarithmic insertion/extraction
- For STL priority_queue, use `top()`, `pop()`, and `push(v)`

```
int values = [1,8,5,6,3,4,0,9,7,2];

priority_queue<int> q;
for(int i = 0; i < 10; ++i)
    q.push(values[i]);
print_queue(q); // A function we write
// 9 8 7 6 5 4 3 2 1 0

priority_queue<int, vector<int>,
greater<int> > q2;
for(int i = 0; i < 10; ++i)
    q2.push(values[i]);
print_queue(q2);
// 0 1 2 3 4 5 6 7 8 9
```

Set, map, priority queue caveats.

For user-defined classes, the `set`, `map`, and `priority_queue` classes all rely on some form of custom comparator to know how to order its elements:

```
struct Comparator {
  // Returns `true` if lhs should be ordered before rhs.
  bool operator() (const Student& lhs, const Student& rhs) const {
    return lhs.get_id() < rhs.get_id();
  }
};
```

Unordered set, map caveats.

- For user-defined classes, the `unordered_set` and `unordered_map` classes rely on some form of hash function to determine its buckets.
- They also need to be able to determine equality between elements.

C++ Minheap

Suppose we want to be able to declare a minheap more easily:

```
    minheap<int> heap;
```

How would you define the minheap type?

C++ Minheap

Using a priority queue:

```
template<typename T>
using minheap = priority_queue<T, vector<T>, greater<T>>;
```

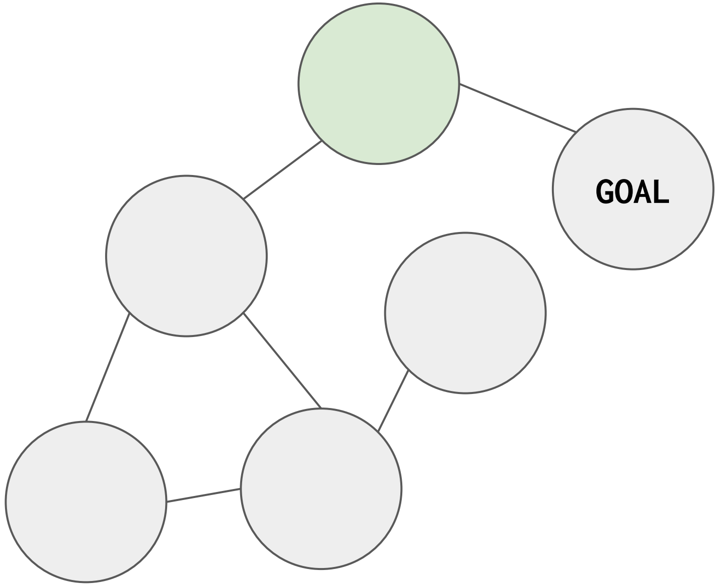
`using` means that we are declaring this type in terms of another type.
`vector<T>` indicates the underlying data structure.
`greater<T>` indicates that **later** values to be popped from the heap will be greater.

Shall we go over Graph Search Strategies (BFS/DFS)?

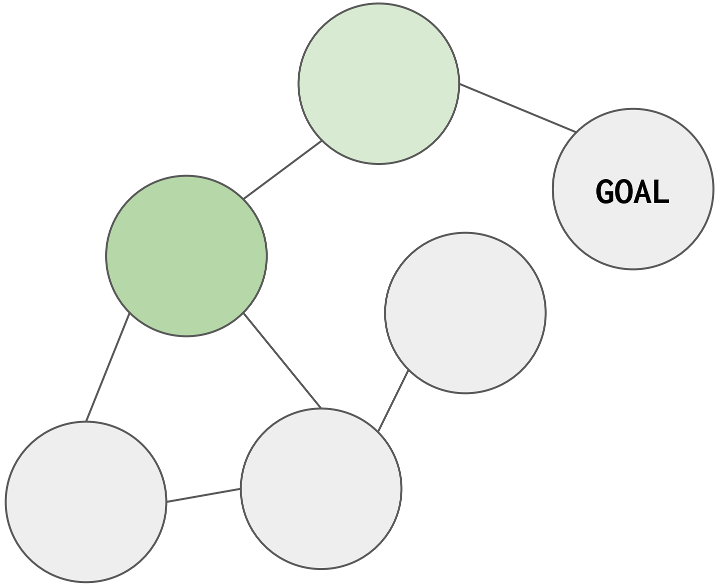
YES

NO

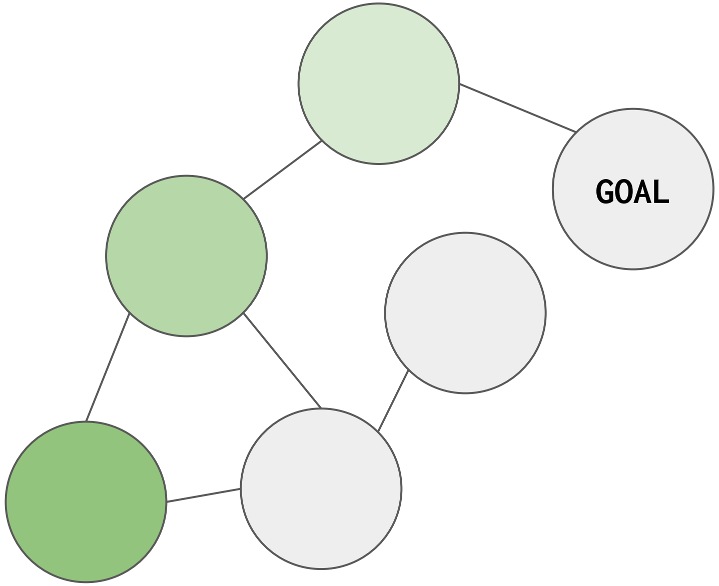
Search strategies: depth-first search



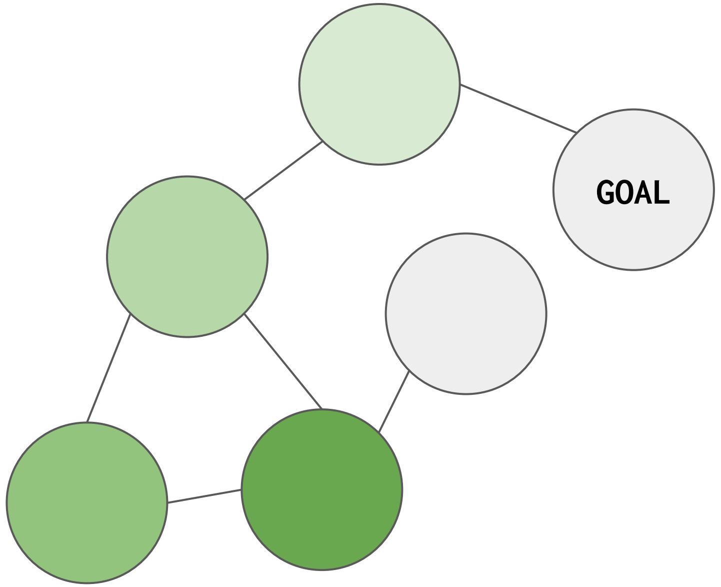
Search strategies: depth-first search



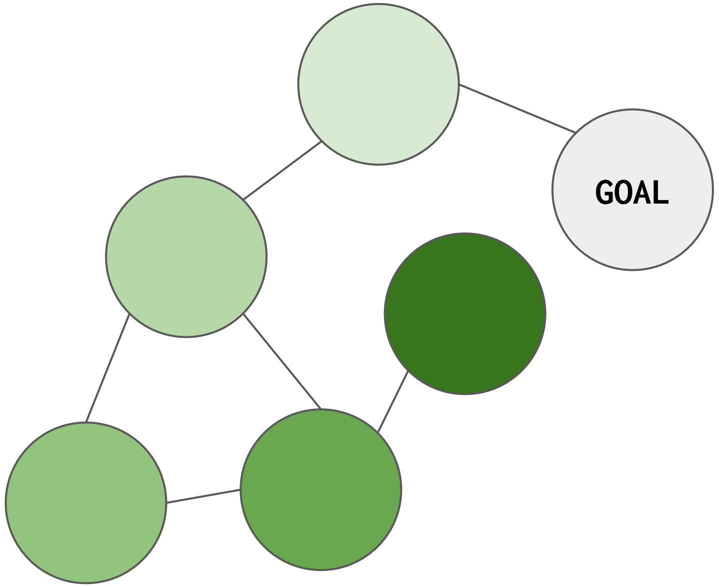
Search strategies: depth-first search



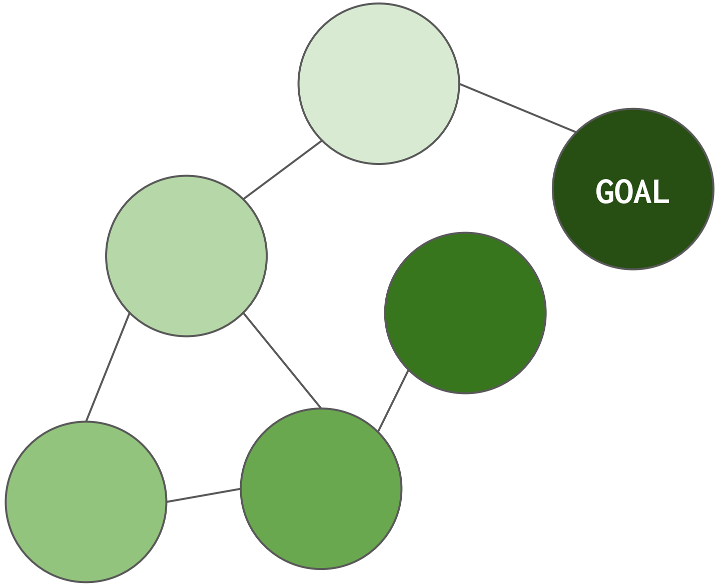
Search strategies: depth-first search



Search strategies: depth-first search



Search strategies: depth-first search



Search strategies: depth-first search

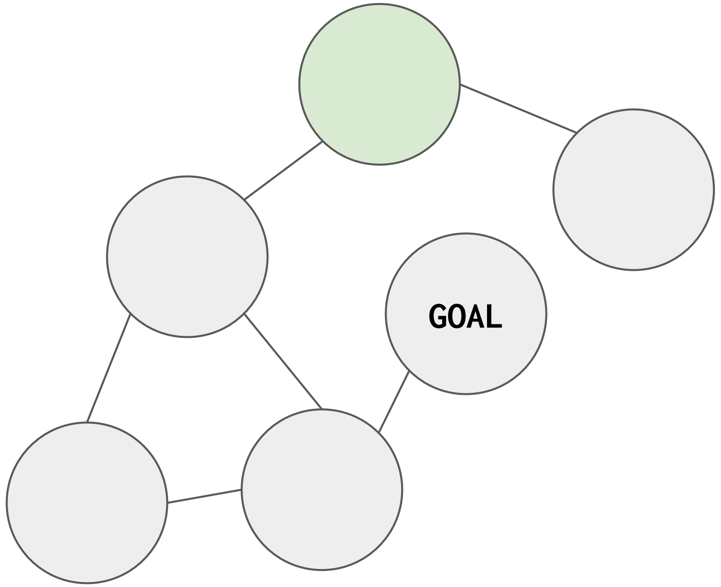
```
Imagine we have an itinerary.
Add our start location to the end of the itinerary.
While our itinerary still has places to visit:
    Grab the last unvisited place we added to the itinerary.
    Move here, and mark this place as visited.
    If this is our goal, we’re done.
    If not, add all places around us to the end of our itinerary.
```

What data structure might be good for this?

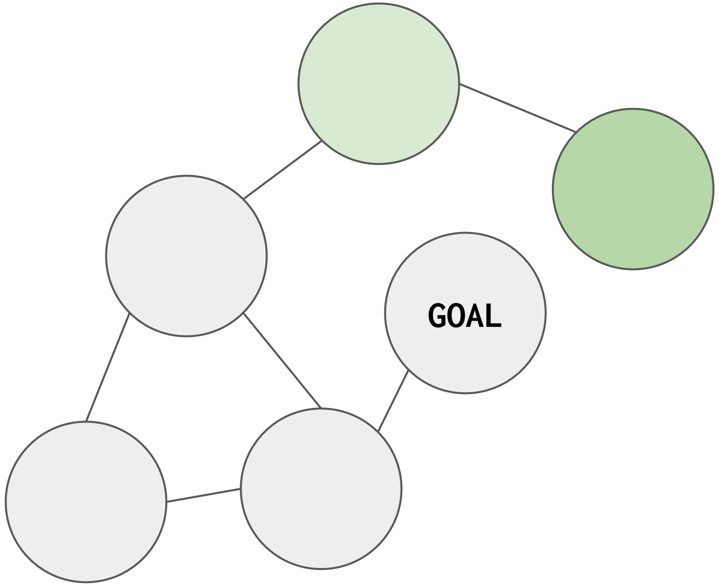
Search strategies: depth-first search

- Depth-first search is *very space efficient!* If you implement it recursively, it only needs to store a single path in memory.
- However, depth-first search is not guaranteed to find the optimal solution.
- Furthermore, depth-first search can potentially fail to terminate in the case that it encounters an infinitely long path.

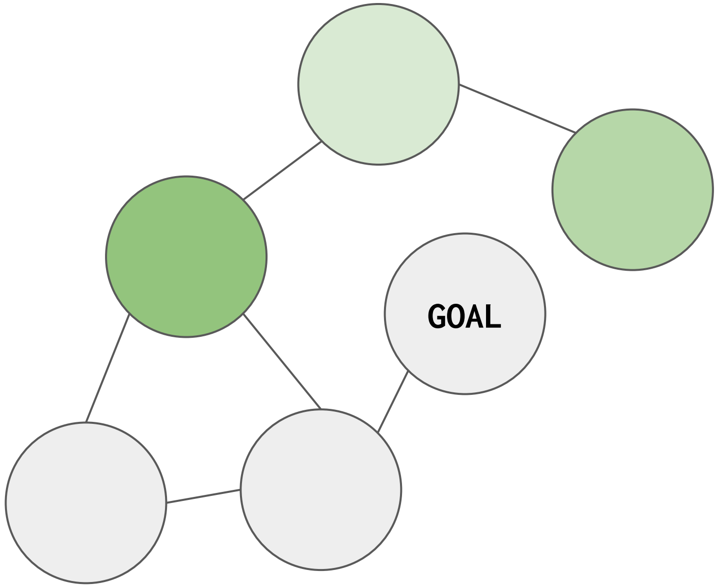
Search strategies: breadth-first search



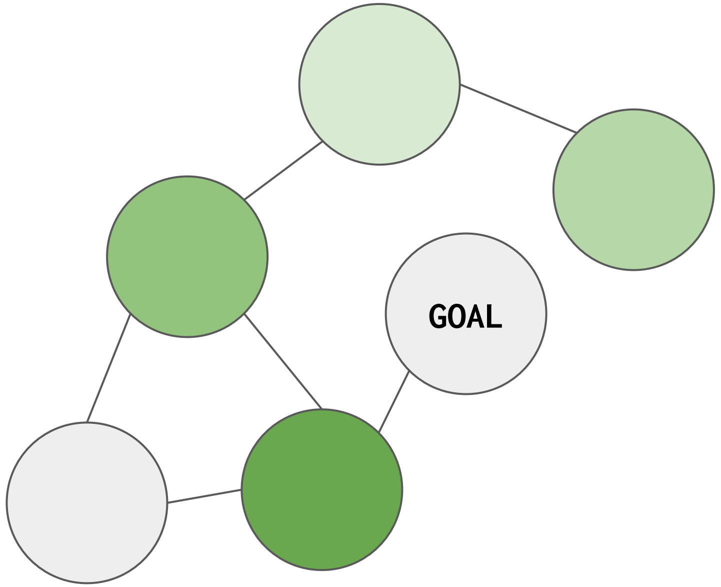
Search strategies: breadth-first search



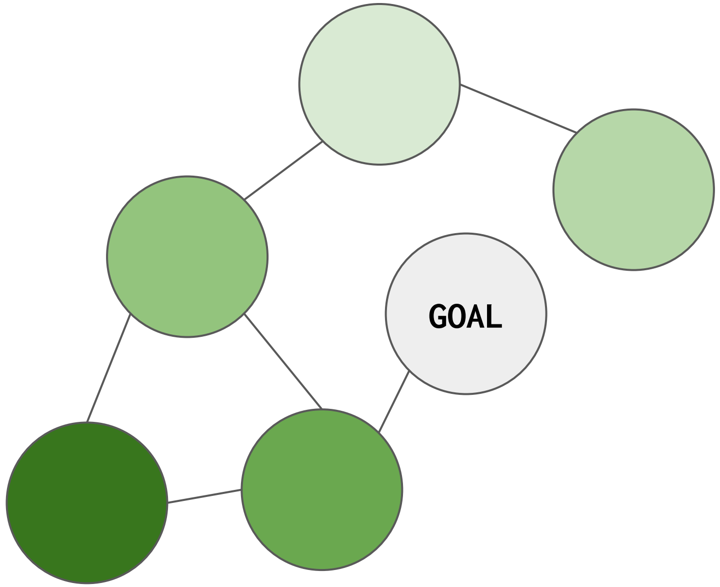
Search strategies: breadth-first search



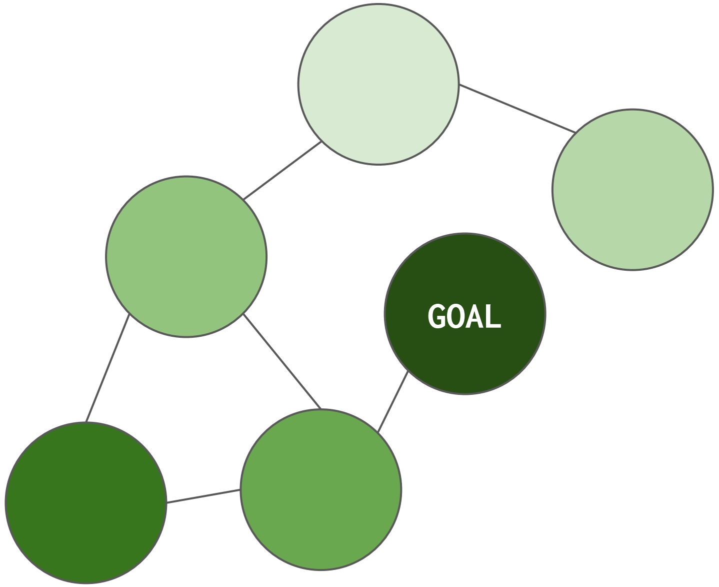
Search strategies: breadth-first search



Search strategies: breadth-first search



Search strategies: breadth-first search



Search strategies: breadth-first search

- We can implement breadth-first search using a queue.
- Unlike the depth-first search, the breadth-first search is guaranteed to find the shortest path.
- Unfortunately, it keeps track of all existing paths, which means that its memory footprint can grow very quickly.

Pseudocode: Depth First Search

```
stack = Stack()
stack.push(newPath(startNode))
seen = Set()
while !stack.isEmpty()
  currPath = stack.pop()
  currState = last(currPath)
  if(currState is goal)
    return currPath
  if(seen contains currState)
    continue
  seen.add(currState)
  for nextState in getNextStates(currState)
    path = newPath(currPath, nextState)
    stack.push(path)
```

Pseudocode: Breadth First Search

```
queue = Queue()
queue.enqueue(newPath(startNode))
seen = Set()
while !queue.isEmpty()
  currPath = queue.dequeue()
  currState = last(currPath)
  if(currState is goal)
    return currPath
  if(seen contains currState)
    continue
  seen.add(currState)
  for nextState in getNextStates(currState)
    path = newPath(currPath, nextState)
    queue.push(path)
```

Good luck!

Sign-in	http://bit.ly/2xeQMKx
Slides	http://bit.ly/2TrABCm
Practice	https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.