Running time

Constant:	$\Theta(1)$
Logarithmic:	$\Theta(\log n)$
Poly-log:	$\Theta\left((\log n)^k\right)$
Linear:	$\Theta(n)$
Log-linear:	$\Theta(n \log n)$
Superlinear:	$\Theta(n^{1+c})$ (c is a constant > 0)
Quadratic:	$\Theta(n^2)$
Cubic:	$\Theta(n^3)$
Polynomial:	$\Theta(n^k)$ (k is a constant) "tractable"
Exponential:	$\Theta(c^n)$ (c is a constant > 1) "intractable"

Suppose you're given the following C++ function. Which of the following is the tightest asymptotic upper bound for the number of lines printed by this function?

```
void fun(int n) {
    for (int b = n * n * n; b > 0; b = b / 3) {
        cout << "I'm being printed!" << endl;
    }
}
```

Let's say this for loop iterates k times

Then $b = b/3^k$

$n^3/3^k \leq 0$ we stop when we said the condition

Since we are using int

$n^3 < 3^k$

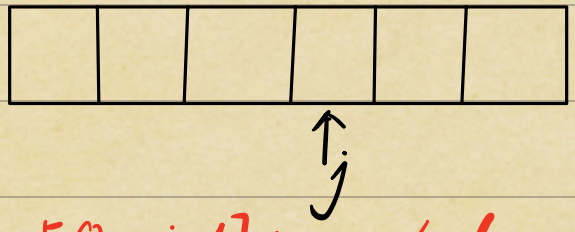
$3\log(n) < k\log(3)$

$k > 3\log(n)/\log(3)$

we can only terminate when we satisfied this condition.

so $O(\log(n))$

Selection Sort



- [0, j-1] is sorted
- [j, n) is unsorted

Time complexity: $O(n^2)$

Space complexity: $O(1)$

```
// Average Time: O(n^2)
// Best TimeL O(n^2)
void selection_sort (vector<string> & candy) {
    // the outter loop runs for n times
    for (int i = 0; i < candy.size(); i++) {
        // findMin runs for n times
        int min = findMin(candy, i);
        // constant time
        swap(candy, min, i);
    }
}
```

Insertion Sort

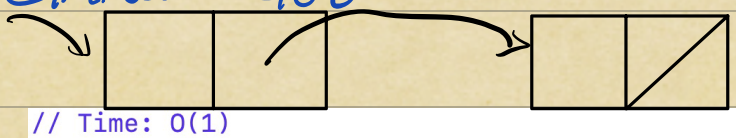
- Iterate from arr[0] to arr[N]
- Compare the current key with its predecessor
- If the key is smaller than its predecessor, compare it to the element before. Move the greater element one position up to make space for swapped element.

```
// Average Time: O(n^2)
// Best Time: O(n)
void insertionSort (vector<string> & candy) {
    // running time is n
    for (int i = 1; i < candy.size(); i++) slide(candy ,i);
}

// running time is n
void slide (vector<string> & candy, int loc) {
    // save the current key on the stack
    string temp = candy[loc];
    int j = loc;
    // if we are not at the beginning and
    // they are still element bigger than the current element
    while (j > 0 && candy[j-1] > temp) {
        // move the larger element one position up
        candy[j] = candy[j-1];
        // go to the previous location
        j--;
    }
    // put the current key into the right position
    candy[j] = temp;
}
```

Time complexity: $O(n^2)$

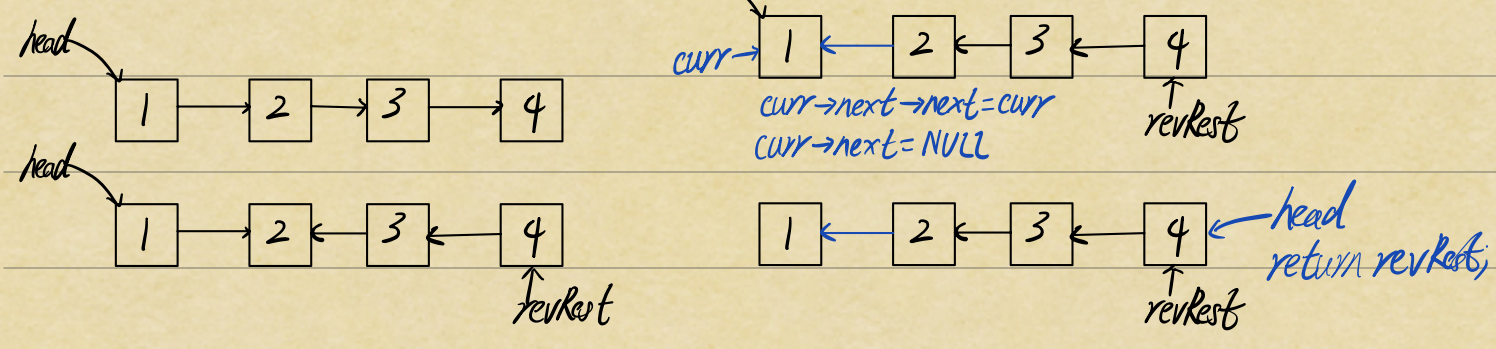
Space complexity: $O(1)$

Linked List



```
// Time: O(1)
// Space: O(1)
void insertAtFront(Node*& curr, LIT e) {
    curr = new Node(e, curr);
}

// Time: O(n)
// Space: O(n)
void printReverse(Node* curr) {
    if (curr != NULL) {
        // go all the way to the bottom, then start printing
        printReverse(curr->next);
        cout << curr->data << end;
    }
}
// Time: O(n)
// Space: O(n)
void printReverseOdds(Node* curr) {
    if (curr == NULL) {
        // if there is no item in the list, return
        return;
    } else if (curr->next == NULL) {
        // if there is only one item in the list, print and return
        cout << curr->data << end;
    } else {
        // if there is more than one item in the list,
        // go to the next layer and then come back to print
        printReverseOdds(curr -> next -> next);
        cout << curr->data << end;
    }
}

// Time: O(n)
// Space: O(n)
Node* reverse(Node* curr) {
    if (curr != NULL && curr->next != NULL) {
        // get the revRest head pointer
        Node* revRest = reverse(curr->next);
        curr->next->next = curr;
        curr->next = NULL;

        // return the head of the reversed list
        curr = revRest;
    }
    return curr;
}
```

Reverse



```
/*
@param head - a pointer to the head of the linked list
@param toDelete - a pointer to the node to be deleted from the list
                  the node is guaranteed to be part of the list
                  this parameter will never be the head or tail of the linked list
*/
void deleteNode(ListNode *&head, ListNode *toDelete) {
    if (head != NULL) {
        while (head != NULL) {
            if (head->value == toDelete->value) {
                ListNode *prevNode = head->prev;
                ListNode *nextNode = head->next;
                nextNode->prev = prevNode;
                prevNode->next = nextNode;
                delete toDelete;
                toDelete = NULL;
                break;  // this is a break statement! it exits the loop.
            }
            head = head->next;
        }
    }
}
```

This code delete the node properly, but it changes the head pointer! So it is wrong.

When implementing doubly linked lists, we often want to use sentinel nodes. What advantage does that give us?

- (a) They make the deletion of arbitrary elements happen in $O(1)$ time.
- (b) They makes the code for deleting the first / last elements the same as for deleting other elements.
- (c) They prevent segmentation faults by warning us when the list is full.
- (d) They allows us to store size data in the sentinel. ✗ sentinel doesn't store this

List

```
// Time: O(n)
// Space: O(n)
Node* Find(Node* place, int k){
    // if we find the index, or if we are at the end, return;
    if ((k == 0) || (place == NULL)) return place;

    // else, go to the next layer
    return Find(place->next, k-1);
}

// Time: O(n)
// Space: O(n)
void list_insert(int loc, LIT e) {
    if (loc > 1) {
        Node* curr = Find(place->next, loc - 1);
        Node* newN = new Node(e);
        newN->next = curr->next;
        curr->next = newN;
    else {
        insertAtFront(head, e);
    }
}
```

```
// Time: O(n)
// Space: O(1)
void removeCurrent(Node*& head, Node*& curr) {
    Node* w = head;
    if (w != curr) {
        // finding the pointer to the current node
        while (w->next != curr) w = w->next;
        w->next = curr->next;
    } else {
        // if this is the head,
        // we just need to update the head pointer
        head = curr->next;
    }

    delete curr;
    curr = NULL;
    size--;
}
```

Stack

```
// ------ Linked List implementation -----
// Time: O(1)
// Space: O(1)
void push(LIT d) {
    Node* newN = new Node(d);
    newN->next = top;
    top = newN;
    size++;
}

// Time: O(1)
// Space: O(1)
LIT pop() {
    // save the return Value
    LIT returnVal = top->data;
    Node* t = top;
    // update top to the second last node
    top = t->next;
    // free the object on the heap
    delete t;
    // update the size of the stack
    size--;
    return returnVal;
}
```

```
// ------ Array implementation ------
// We are only modifing the right side of the vector
// Time: O(1)
// Space: O(1)
void push(LIT d) {
    items.push_back(d);
}

// Time: O(1) per push if we scale up by c every time
// Space: O(1)
LIT pop() {
    // peek for the return value
    LIT returnVal = items[items.size - 1];
    // pop the last one out
    items.pop_back();
    return returnVal;
}
```

Merge sort

```
// Average Time: O(n*logn)
// Best Time: O(n)
void mergeSort(vector<T>& A, int lo, int hi) {
    // if the array has 0 or 1 elements, it is sorted, stop
    if (hi > lo) {
        // split the array to two different size
        int mid = (hi + lo)/2;
        // sort the lower part
        mergeSort(A, lo, mid);
        // sort the high part
        mergeSort(A, mid + 1, hi);
        // merge the sorted halves to produce one sorted result
        // Time: O(n)
        merge(A, lo, mid, hi);
    }
}
```

$T(N) = T(N/2) \cdot 2 + n$

Which of the following is true about performing merge sort using linked lists (rather than arrays)?

- ○ (a) The algorithm can be performed in (otherwise) worst-case $O(log(n))$ time if the two halves it breaks the input into are already sorted.
- ○ (b) The algorithm can run with worst-case space complexity $O(1)$ (i.e., using $O(1)$ additional space beyond what was already used for the input) X we need O(logn) space for call stack
- ◉ (c) None of the other answers are true
- ○ (d) The algorithm can only be performed on doubly linked lists, not singly linked lists.
- ○ (e) The algorithm can be done in worst-case $O(n)$ time

Queue

```
// ------ Singly Linked List implementation with head and tail pointer------
// head is entry, tail is exit
// Time: O(1)
// Space: O(1)
void enqueue(LIT d) {
    Node* newN = new Node(d);
    newN->next = head;
    head = newN;
    size++;
}

// head is entry, tail is exit
// Time: O(n)
// Space: O(1)
LIT dequeue() {
    // save the return Value
    LIT returnVal = tail->data;
    Node* temp = tail;
    // update the tail pointer to its previous value
    tail = findBefore(tail);
    delete temp;
    size--;
    return returnVal;
}
```

```
// ------ Using two stacks ------
public class Queue<E>
{
    private Stack<E> inbox = new Stack<E>();
    private Stack<E> outbox = new Stack<E>();

    // always push to the inbox
    // Time: O(1)
    public void queue(E item) {
        inbox.push(item);
    }

    // always pop from teh outbox
    // Time: average O(1) per operation
    public E dequeue() {
        // when the outbox is empty, refill
    // O(n) for an operation
        if (outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
               outbox.push(inbox.pop());
            }
        }
    // however, on average of n operation, this function will be constant
        return outbox.pop();
    }
}
```

```
// head is exit, tail is entry
// Time: O(1)
// Space: O(1)
void enqueue(LIT d) {
    Node* newN = new Node(d);
    newN->next = tail->next;
    tail->next = newN;
    tail = newN;
    size++;
}

// head is exit, tail is entry
// Time: O(1)
// Space: O(1)
LIT dequeue() {
    // save the return Value
    LIT returnVal = head->data;
    Node* temp = head;
    // update the head pointer to its next value
    head = head->next;
    delete temp;
    size--;
    return returnVal;
}
```

Recursive Definition

Which recursive definition best describes a problem in which each step divides the problem into four problems of one fourth (1/4) the size, where the dividing and merging each take time proportional to the size of the problem?

$T(n) = 4\,T(n/4) + n$

↑ merging time

Let $T(n) = 2T(\frac{n}{2}) + 5, T(1) = 1$. Assume n is a power of 2.

What is a closed-form solution to this recurrence?

- (a) $T(n) = 5(\frac{n}{2}) + \log_2(n)$
- (b) $T(n) = 5\log_2(n) + 1$
- (c) $T(n) = 2(\frac{n}{2}) + 5$
- (d) $T(n) = 6n - 5$

Assume $T(k) = 6k-5$ for all $k<n$

$T(n) = 2T(n/2)+5$

$= 2(6(n/2)-5)+5$ by inductive hypothesis

$= 6n-5$

Trees

Branching for d-ary tree:

Full d-ary tree: every node has 0 or d children

Perfect d-ary tree: has as many node as possible for a given height

Complete d-ary tree: a perfect tree with nodes on the lowest level from right removed

complete perfect

$2^h \le n \le 2^{h+1} - 1$

Full tree: $n \ge 2h+1$

$n-1/2 \ge h$

$\log_2(n+1) - 1 \le h \le n-1$

perfect complete

every node only has one child

N item binary tree has N+1 null pointers.

```
// Time: O(n)
void clear(Node*& croot) {
    if (croot != NULL) {
        clear(croot->left);
        clear(croot->right);
        delete croot;
        croot = NULL;
    }
}

// Time: O(n)
Node* copy(Node* croot) {
    Node* temp = NULL;
    if (croot != NULL) {
        temp = new Node(croot->data);
        temp->left = copy(croot->left);
        temp->right = copy(croot->right);
    }
    return temp;
}
```

post order

pre order

A **complete** binary tree with 273 nodes, has integer leaves on the deepest level.

Find the full tree with $h-1$ first

$Nf = 2^{(h-1)+1} - 1 = 2^h - 1$

Find height

$\log_2(273+1) - 1 = 7.098 \leq h$

$h = 8$,

$Nf = 2^8 - 1 = 255$

$Nl = 273 - 255 = 18$

A **perfect** binary tree with height 7, has integer ? leaves.

Find total: $N = 2^{(7+1)} - 1$

Find non-leaves: $N_n = 2^{(7-1+1)} - 1$

$N_l = 2^8 - 2^7 = 2^7(2-1) = 2^7$

$N_l = 2^h$

Traversal

We can determine the level-order traversal of a binary tree is we are given either the pre-order traversal and the in order traversal or the post-order traversal and the in order traversal.

Level Order: $O(n)$

Pre-Order: $O(n)$

```
// Time: O(n)
void levelOrder(Node* croot) {
    queue<Node*> Q;
    Q.push(croot);
    while (!Q.isempty*()) {
        Node* t = Q.front();
        Q.pop();
        // if this node is not NULL, enqueue all its child
        if (t) {
            yell t;
            Q.push(t->left);
            Q.push(t->right);
        }
    }
}
```

Binary Search Tree

The left subtree contains only nodes with keys less than the node's key.

The right subtree contains only nodes with keys larger than the node's key.

The left and right subtree must also be a BST.

Use in-order traversal for sorted key.

```
// Time: O(h)
void insert(Node*& croot, const K & key) {
    if (croot == NULL) {
        croot = new Node(key);
    } else if (key < croot->key) {
        insert(croot->left);
    } else if (key > croot->key) {
        insert(croot->right);
    }
    // if key == croot->key, nothing happens, since we want to maintain uniqueness
}

// Time: O(h)
Node*& find(K & key, Node*& r) {
    if (r == NULL) return NULL;
    else if (r->key == key) return r;
    else if (r->key < key) find(r->left, key);
    else find(r->right, key);
}

// Time: O(h)
void remove(Node*& croot, const K & k) {
    if (croot != NULL) {
        if (croot->key == key) {
            doRemoval(croot);
        } else if (k < croot->key) {
            remove(croot->left, k);
        } else {
            remove(croot->right, k);
        }
    }
}
// Time: O(h)
void doRemoval(Node*& croot) {
    if ((croot->left != NULL) && (croot->left != NULL)) {
        // if there are two child, we need to maintain the in order traversal
        twoChildRemove(croot);
    } else {
        // if there is only one child, we just need to fix the pointer
        zeroOneChildRemove(croot);
    }
}

void zeroOneChildRemove(Node*& croot) {
    Node* temp = croot;
    if (croot->left == NULL) croot = croot->right;
    else croot = croot->left;
    delete temp;
}

void twoChildRemove(Node*& croot) {
    // find the right most child of the left subtree
    Node*& iop = rightMostChild(croot->left);
    // change the key in the croot
    croot->key = iop->key;
    // remove this child to avoid duplicate
    zeroOneChildRemove(iop);
}

Node*& rightMostChild(Node*& root){ find in order predesessor
    // if there is no more right child, return this root
    if (root->right == NULL) return root;
    // if we are not yet at the end, continue finding
    else return rightMostChild(root->right);
}
```

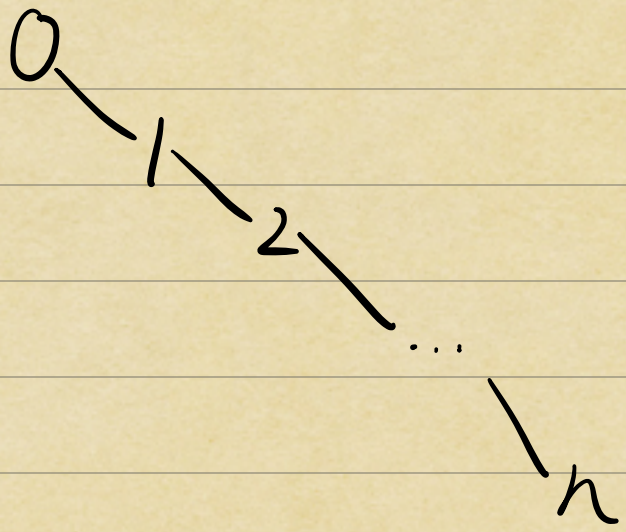
we can also do removal by finding the left most child of the right subtree

Choose the appropriate running time from the list below.

The variable n represents the number of items (keys, data, or key/data pairs) in the structure. In answering this question you should assume the best possible implementation given the constraints, and also assume that every array is sufficiently large to handle all items (unless otherwise stated).

Build a BST with keys that are the numbers between 0 and n, in that order, by repeated insertions into the tree.

- (a) $O(n)$
- (b) $O(1)$
- (c) $O(n^2)$
- (d) $O(\log n)$
- (e) $O(n \log n)$



This BST only has right children. Takes O(n) time to find a position for each insertion.
For n times, $n \cdot O(n) = O(n^2)$

Choose the appropriate running time from the list below.

The variable n represents the number of items (keys, data, or key/data pairs) in the structure. In answering this question you should assume the best possible implementation given the constraints, and also assume that every array is sufficiently large to handle all items (unless otherwise stated).

Compute the height of every subtree in a Binary Search Tree.

- (a) $O(\log n)$
- (b) $O(1)$
- (c) $O(n \log n)$
- (d) $O(n)$
- (e) $O(n^2)$

We need to visit all the nodes to determine the height.

Consider a BST with n nodes. What is the minimum and maximum number of comparisons that you need to make in order to insert a new node?

- (a) We do not have enough information to answer the question.
- (b) $\log n$ and n
- (c) 1 and n
- (d) $\log n$ and $\log n$
- (e) 1 and $\log n$

The maximum height is $n-1$.

ADT Dictionary

	insert	find	(findt) remove
Linked list	$\Theta(n)+\Theta(1)$ anywhere (uniqueness)	$\Theta(n)$	$\Theta(n)+\Theta(1)$
Unsorted array	$\Theta(n)+\Theta(1)$ to the end	$\Theta(n)$	$\Theta(n)+\Theta(1)$
Sorted array	$\Theta(\log n)+\Theta(n)$ (binary search, copy to back)	$\Theta(\log n)$	$\Theta(\log n)+\Theta(n)$
Binary Search Tree	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

AVL Tree

Rebalance:

```
If height(left)-height(right)> 1:
    If height(left_left) >= height(left_right):
        rotateRight
    else:
        rotateLeftRight
else if height(left)-height(right)<-1
    If height(right_right) >= height(right_left):
        rotateLeft
    else:
        rotateRightLeft
```

```
template <class K, class V>
void AVLTree<K, V>::rotateLeft(Node*& t) {
    *_out << __func__ << endl; // Outputs the rotation name (don't remove this)

    Node* newSubRoot = t->right;
    t->right = newSubRoot->left;
    newSubRoot->left = t;
    t = newSubRoot;

    // update the heights for t->left and t (in that order)
    updateHeight(t->left);
    updateHeight(t);
}

template <class K, class V>
void AVLTree<K, V>::rotateRight(Node*& t) {
    *_out << __func__ << endl; // Outputs the rotation name (don't remove this)

    Node* newSubRoot = t->left;
    t->left = newSubRoot->right;
    newSubRoot->right = t;
    t = newSubRoot;

    // updates heights
    updateHeight(t->right);
    updateHeight(t);
}
template <class K, class V>
void AVLTree<K, V>::rotateLeftRight(Node*& t) {
    *_out << __func__ << endl; // Outputs the rotation name (don't remove this)

    rotateLeft(t->left);
    rotateRight(t);
}

template <class K, class V>
void AVLTree<K, V>::rotateRightLeft(Node*& t) {
    *_out << __func__ << endl; // Outputs the rotation name (don't remove this)

    rotateRight(t->right);
    rotateLeft(t);
}
template <class K, class V>
void AVLTree<K, V>::rebalance(Node*& subtree) {
    // positive values mean left is "heavier", negative means right is "heavier"
    int balance = height(subtree->left) - height(subtree->right); // checks which side is "heavier"

    if (balance > 1) { // if left side larger than right
        if (height(subtree->left->left) >= height(subtree->left->right)) {
            rotateRight(subtree);
        } else {
            rotateLeftRight(subtree);
        }
    } else if (balance < -1) { // if right side larger than left
        if (height (subtree->right->right) >= height(subtree->right->left)) {
            rotateLeft(subtree);
        } else {
            rotateRightLeft(subtree);
        }
    } else {
        updateHeight(subtree); // if balanced
    }
}
```

Suppose we have an AVL tree of height 5. What is the minimum number of nodes that could be in **left subtree of the root?**

$N(0)=1$ # base case

$N(1)=2$ # base case

$N(h)=N(h-1)+N(h-2)+1$

Since the height of tree is 5, left height of root can be 4.

So, the height of the left subroot is 3.

$N(3)=N(2)+N(1)+1$

$=N(1)+N(0)+N(1)+1+1$

$=7$

B-Tree

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a boolean value $x.leaf$ which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most $n-1$ keys along with a pointer to each child.
4. Each node except root can have $[\lceil n/2 \rceil, n]$ children. root: $[2, n]$
5. The root has at least 2 children and contains at least 1 key.
6. If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $\log_t \frac{n+1}{2} \geq h \geq \log_t (n+1)/2$

$1+2(t-1) \le keys \le n^{h+1}-1$ $\quad node \le \frac{m^{h+1}-1}{m-1}$

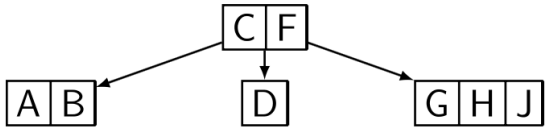
What is the minimum number of keys that can be stored in a B-Tree of order 64 and height 5?

- (a) $2^{25}+1$
- (b) $2^{25}-1$
- (c) $2^{26}-1$
- (d) $2^{30}+1$
- (e) $2^{30}-1$

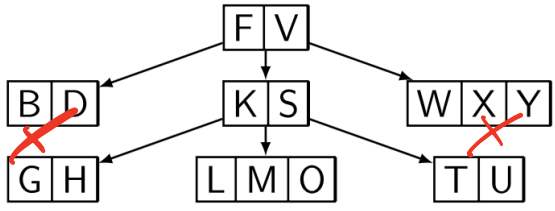
In B-tree of order 64, the minimum degree is $64/2 = 32$.
Then min num of key is $(32^5-1)\cdot 2+1 = 2^{26}-1$

Which of these two trees are valid B-Trees of order 4?

1.



2.



If you are not the last level, you must have a child.

- (a) Neither (1) nor (2) is valid.
- (b) Only (1) is valid.
- (c) Both (1) and (2) are valid.
- (d) Only (2) is valid.

Which of the following statements is false for a B-tree of order m containing n items?

(i) The height of the B-tree is $O(\log_m n)$.
(ii) A node contains a maximum of $m-1$ keys, and this is an upper bound on the number of key comparisons at each level of the tree during a search.
(iii) For fixed n, decreasing m increases the number of disk seeks.

- (a) At least two of (i), (ii) and (iii) are false.
- (b) None of these characteristics is false.
- (c) Only (ii) is false.
- (d) Only (iii) is false.
- (e) Only (i) is false.

Which of the following statements is true for a B-tree of order m containing n items?

(i) The height of the B-tree is $O(\log_m n)$ and this bounds the total number of disk seeks.
(ii) A node contains a maximum of $m - 1$ keys, and this bounds the number of disk seeks at each level of the tree.
(iii) Every Binary Search Tree (or AVL tree) is also an order 1 B-Tree.

- (a) Two of the statements are true.
- (b) None of the statements are true.
- (c) Only item (iii) is true.
- (d) Only item (ii) is true.
- (e) Only item (i) is true.

Hashing

1) Separate Hashing Open hashing

Chain keys using a linked list when conflict occurs.

2) Open addressing Closed hashing

a) Linear Probing

1. Calculate the hash key
2. Check if hashTable[key] is empty
   store the value directly by hashTable[key] = data
3. If the hash index already has some value then
   check for the next index using key = (key+1)%size
4. Check, if the next index is available hashTable[key]
   then store the value. Otherwise try again.

b) Quadratic Probing

1. If the slot hash(x)%n is full, try (hash(x)+1²)%n
2. Then try (hash(x)+2²)%n
3. Then try (hash(x)+3²)%n

c) Double Hashing

1. The first hashing function takes the key and gives out a location on the hash table.

2. If occupied, we will use this:

$h(k,i) = (h_1(k) + i \times h_2(k)) \% n$

i indicates collision number

k key

n size of table

Time: O(n)

The CS department wants to maintain a database of up to 2900 student numbers of students who have taken CS 221 so that it can be determined very quickly whether or not a given student has taken the course. Speed of response is very important; efficient use of memory is also important, but not as important as speed of response. To be more specific, we would like to be able to do our lookup in $O(1)$ time, and use a reasonable amount of memory to do so. Which of the following data structures would be most appropriate for this task?

- (a) A hash table using probing with capacity 6,000
- (b) A sorted linked list
- (c) A hash table using probing with capacity 2,900
- (d) A hash table using probing with capacity 20,000
- (e) A sorted array with 2,900 entries

larger space means less conflict, and too much space is bad.

Suppose we are given two hash functions, $h_1(k)$ and $h_2(k)$ which hash values k as follows:

k	$h_1(k)$	$h_2(k)$
0	1	1
1	2	1
2	3	2
3	4	3
4	4	4
5	3	5
6	2	6
7	1	7

Suppose we use $h_1(k)$ as our primary hash function, but want to resolve collisions using the double hashing technique with $h_2(k)$. Given a 0-indexed hash table with `size = 9`, suppose we hash the following values to our hash table in the following order:

`1 0 3 4 6 2`

Given the information above, if we were to print out the value of our hash values by increasing index, ignoring any empty entries, what order would these values be printed?

$h(x,i) = (h_1(k) + i \times h_2(k)) \% n$

0	
1	0
2	1
3	2
4	3
5	6
6	
7	
8	4

h(1,0) = (2+0)%9 = 2 no

h(0,0) = (1+0)%9 = 1 no

h(3,0) = (4+0)%9 = 4 no

h(4,0) = (4+0)%9 = 4 Yes

h(4,1) = (4+1·4)%9 = 8 no

h(6,0) = (2+0)%9 = 2 Yes

h(6,1) = (2+1·6)%9 = 8 Yes

h(6,2) = (2+2·6)%9 = 5 no

h(2,0) = (3+0)%9 = 3 no

A hash table of length 10 uses open addressing with hash function $h(k) = k \mod 10$, and linear probing.

Index	Value
0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

This is the state of the hash table after inserting 6 values. Which of the following gives a correct order in which values might have been inserted into the hash table?

- (a) More than one of the insertion orders are correct
- (b) 46, 42, 34, 52, 23, 33
- (c) 46, 34, 42, 23, 52, 33
- (d) 34, 42, 23, 52, 33, 46

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

46 % 10 = 6 no

34 % 10 = 4 no

42 % 10 = 2 no

23 % 10 = 3 no

52 % 10 = 2 Yes (52+1)%10 = 3 Yes (52+2)%10 = 4 Yes

(52+3)%10=5 no

33%10=3 Yes (33+4)%10=7 no

Given a hash table `T` that holds 4000 elements and has 25 slots, the load factor `alpha` for `T` is:

- (a) 2.50
- (b) 160
- (c) None of the other options are correct.
- (d) 160000
- (e) 0.0250

Load Factor = Total elements in hash table / size of hash table

$\alpha = 4000/25 = 160$

You want to build an efficient spell-checker application for a Microsoft Word document made up of 10000 words. What type of collision resolution would you adopt if your hash function generates indexes based on the first character of the word, that is, for the word `apple` the output is 0, for the word `banana` the output is 1 and so on.

- (a) Linked List
- (b) Either of Linear Probing and Separate Chaining can be used
- (c) Linear Probing
- (d) None of the others
- (e) Separate Chaining

A lot of words start with same character.

Which of the following statement(s) are correct about collision?

i) Two entries are identical except for their keys.

ii) Two entries with different data have the exact same key.

iii) Two entries with different keys have the same exact hash value.

iv) Two entries with the exact same key have different hash values.

- (a) ii and iii only
- (b) i and iii only
- (c) iii only
- (d) iV only
- (e) i only

What condition of the following indicates non-determinism in hash procedure?

- (a) Two entries are identical except for their keys.
- (b) Two entries with different keys have the same exact hash value.
- (c) Two entries with the exact same key have different hash values.
- (d) Two entries with different data have the exact same key.
- (e) None of the other options are correct.

Which of the following statement(s) is TRUE?

(i) A hash function takes a message of arbitrary length and generates a fixed length code.

(ii) A hash function takes a message of fixed length and generates a code of variable length.

(iii) A hash function may give the same hash value for distinct messages.

- (a) ii and iii only
- (b) i only
- (c) ii only
- (d) i and iii only
- (e) None of the others options are correct.

An ideal hashing function is:

1. deterministic always produce the same hash value for the same input value
2. distributes keys uniformly
3. computed in constant time

Heap

Max-Heap: The value of the root node must be greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.

Min-Heap: The value of the root node must be smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.

Indexing:

	$i \geq 0$	$i \geq 1$
Left child:	$2 \cdot i + 1$	$2i$
Right child:	$2 \cdot i + 2$	$2i + 1$
Parent:	$(i-1)/2$	$i/2$

Level order traversal for sorted array

```
// ------ Min-Heap implementation ------
// For some reason, we use index 1 as the root
// and we use size as the last index of the array

int leftChild(int parent) {
    return 2 * parent;
}

int rightChild(int parent) {
    return 2 * parent + 1;
}

int parent(int child) {
    return child / 2;
}

bool hasAChild(int parent) {
    return leftChild(parent) <= size;
}

int minChild(int parent) {
    if (rightChild(parent) <= size)
        return items[leftChild(parent)] <= items[rightChild(parent)]
            ? leftChild(parent) : rightChild(parent);
    else
        return leftChild(parent);
}

void swap(T& left, T& right) {
    T temp = left;
    left = right;
    right = temp;
}
```

```
// Time: O(log(n)) = O(h)
void heapifyDown(int curr) {
    // find if this curr has a child
    if (hasAChild(curr)) {
        minChildIndex = minChild(curr);
        if (items[curr] > items[minChildIndex]) {
            swap(items[curr], items[minChildIndex]);
            heapifyDown(minChildIndex);
        }
    }
}
```

```
// Time: O(log(n)) = O(h)
void insert(const T & key) {
    if (size == capacity) growAway();
    size+;
    items[size] = key;
    heapifyUp(size);
}

// Time: O(log(n)) = O(h)
void heapifyUp(int curr) {
    if (curr > 1) {
        // if we are not at the root
        if (items[curr] < items[parent(curr)]) {
            // if the child is smaller than the parent, we swap
            swap(items[curr], items[parent(curr)]);
            // go up one layer to check once again
            heapifyUp(parent(curr));
        }
    }
}

T removeMin() {
    // give the root to the user
    T minVal = items[1];
    // pick the leaf to the root
    items[1] = items[size];
    size--; // remove the leaf fromt the array
    heapifyDown(int curr);
    return minVal;
}
```

```
// Time: O(nlog(n))
// incremented insertion
void buildHeap() {
    for (int i = 2; i <= size; i++)
        heapifyUp(i);
}

// Time: O(n)
// decremented insertion
void buildHeap() {
    for (int i = parent(size); i > 0; i--)
        heapifyDown(i);
}
```

Best One!

Fill in the blanks: For a perfect tree of height h containing $n = 2^{h+1} - 1$ nodes, an efficient implementation of `BuildHeap` will call _____ at most _____ times.

- (a) `HeapifyUp`, n
- (b) `HeapifyUp`, $n/2$
- (c) `HeapifyDown`, $n/2$
- (d) `HeapifyDown`, n

What characteristic of Heaps allow them to be stored efficiently in an array?

- (a) Heaps are perfect trees.
- (b) Heaps are binary trees.
- (c) Heaps contain comparable keys.
- (d) Heaps are complete trees.
- (e) None of the other choices is a sufficient explanation.

Disjoint set

```
// ------ Up Tree implementation ------
// the index if the variable,
// which the value is the parent of curr

// Average Time: O(log(n))
// Worst Time: O(n)
int Find(int i) {
    // if the value at i is negative, we are at the root
    if (s[i] < 0) return [i];
    // else, go up once again to find the root
    else return Find(s[i]);
}

// Time: O(1)
int Union(int root1, int root2) {
    // make the root of root2 to be root 1;
    s[root2] = root1;
}

// Time: O(1)
// This one ensure the height of the tree is log(n)
int UnionBySize(int root1, int root2) {
    // the root has the negative value of its height
    int newSize = s[root1] + s[root2];
    if (isBigger(root1, root2)) {
        // if the size of set1 is larger than set2, set2 points to set1
        s[root2] = root1;
        root1 = newSize;
    } else {
        // same if opposite happen
        s[root1] = root2;
        root2 = newSize;
    }
}
```

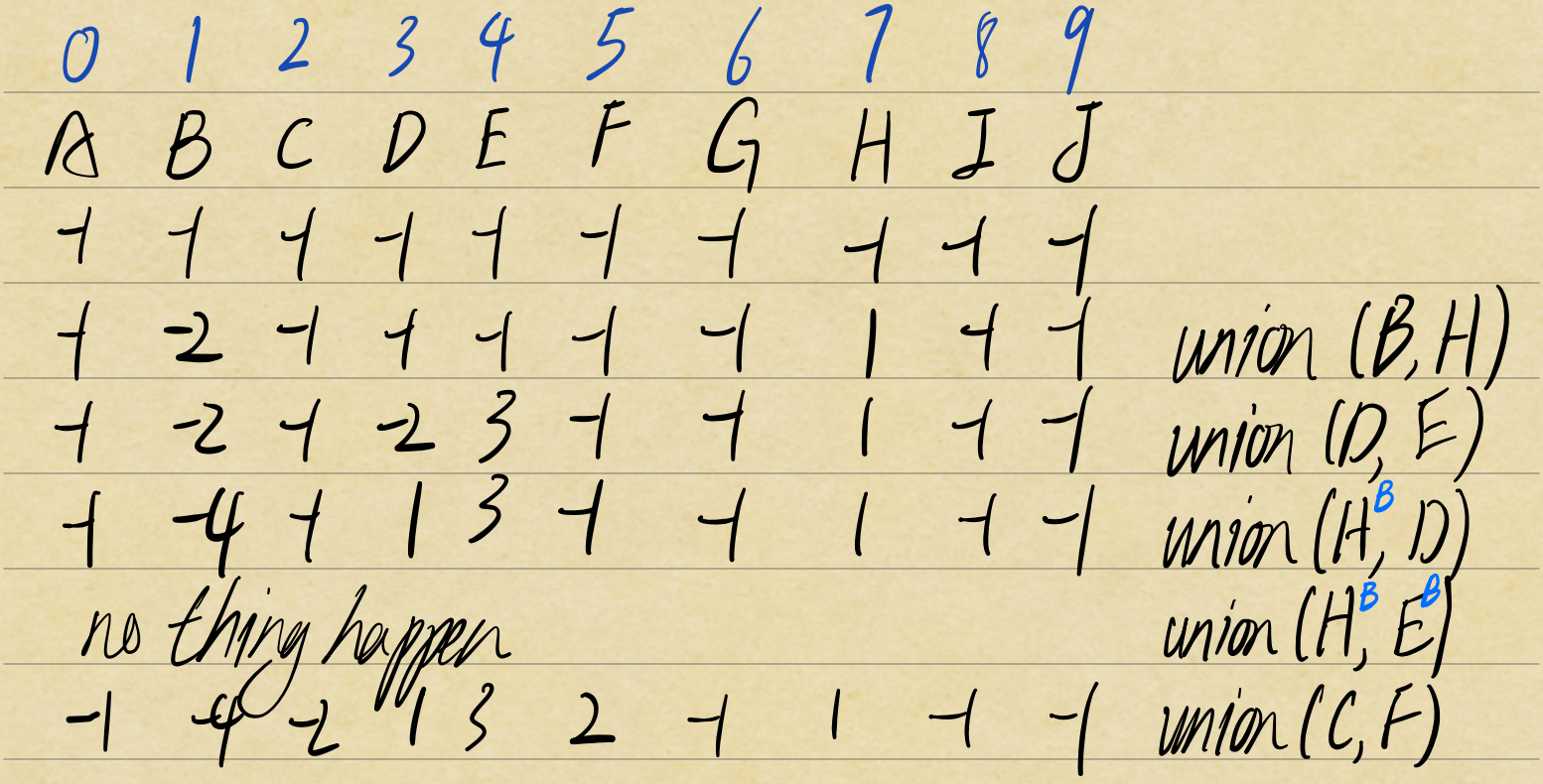
For a disjoint set structure holding n elements, what is the best runtime for find, on average, over a sequence of m operations?

- (a) $O(m)$
- (b) $O(m \log^*(n))$
- (c) $O(m \log(n))$
- (d) $O(m \log(\log(n)))$

We are given an array representation of a disjoint set, where the array value for a root is the negative of its tree's size. There are 10 elements, A through J, where A is given index 0, B is given index 1, ... , and J is given index 9. What is the resulting array after the following union operations are performed? Assume **smart-union by size**. In the event that the two sizes are equal, the first argument will be considered larger than the second. Assume that find **does not use path compression**

```
union(B,H)
union(D,E)
union(H,D)
union(H,E)
union(C,F)
```

- (a) [-1, 7, 5, 4, -4, -2, -1, 4, -1, -1]
- (b) [-1, 7, -2, 7, 7, 2, -1, -4, -1, -1]
- (c) [-1, -4, -2, 1, 1, 2, -1, 1, -1, -1]
- (d) [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
- (e) [-1, -4, -2, 1, 3, 2, -1, 1, -1, -1]



Suppose you are given the following array based representation of a forest of uptrees. The array value for a root is minus the size of its tree. The forest originally consisted of 10 single element sets, but over the structure's lifetime it has changed.

	0	1	2	3	4	5	6	7	8	9
arr	3	3	3	-9	1	3	3	-1	3	8

We say that a Union operation "succeeds" if it combines two originally different sets, or that it "fails" if both arguments to the Union operation are the same set.

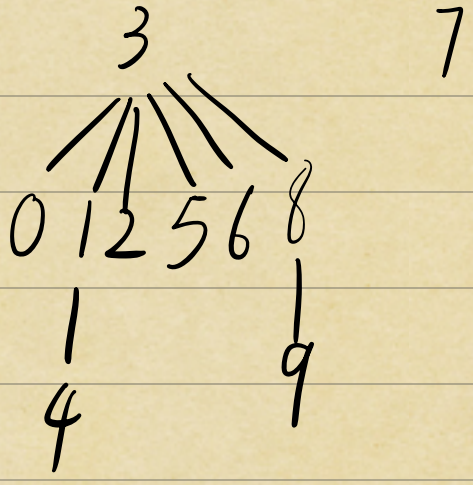
How many successful Union operations must have occurred in the history of the structure above? integer 8

What is the height of the tallest uptree in the example above?

integer 2

Suppose that over the lifetime of a Disjoint Sets structure holding 472 elements there are 397 successful union operations. How many uptrees exist in the structure?

integer $472-397=75$



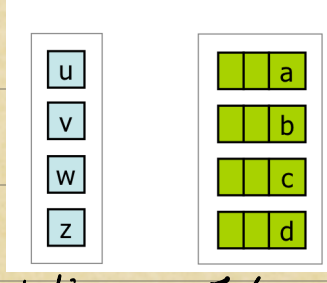
Graph

Connected subgraph: $\exists$ maximal connected subgraph

Spanning Tree: connected, acyclic subgraph, containing all vertices

$|V|-1 \leq |E| \leq \frac{|V| \cdot |V-1|}{2}$

$|V| \cdot D/2 \notin \mathbb{Z}$ not possible



Adjacency Matrix

Edge List

	u	v	w	z
u				
v				
w				
z				

Adjacency Matrix

• n vertices, m edges • no parallel edges • no self-loops • bounds are big-Θ	Edge List	Adjacency List	Adjacency Matrix
Space	$n+m$	$n+m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent(v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n ~~n^2~~
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n ~~n^2~~
removeEdge(e)	1	1	1

Suppose we run depth first search on a connected, undirected graph with n vertices and 5n + 2 edges. How many edges will be labelled "back" edges?

- (a) $5n + 2$
- (b) $5n + 4$
- (c) $4n + 1$
- (d) $n - 1$
- (e) $4n + 3$

The total number of edges in DFS is n-1.

5n+2-(n-1) = 4n+3

BFS

```
Algorithm BFS(G,v)
    Input: graph G and start vertex v
    Output:  labeling of the edges of G in the connected component of v as discovery edges and cross edges
queue q;
setLabel(v, VISITED)
q.enqueue(v);
While !(q.isEmpty)
  q.dequeue(v)
  For all w in G.adjacentVertices(v)
     if getLabel(w) = UNEXPLORED
        setLabel((v,w),DISCOVERY)
        setLabel(w, VISITED)
        q.enqueue(w)
     else if getLabel((v,w)) = UNEXPLORED
        setLabel((v,w),CROSS)
```

P[w]=V
d[w]=1+d[v]

!

```
Algorithm BFS(G)
    Input: graph G
    Output:  labeling of the edges of G as discovery edges and back edges

For all u in G.vertices()
  setLabel(u, UNEXPLORED)
For all e in G.edges()
  setLabel(e, UNEXPLORED)

For all v in G.vertices()
  if getLabel(v) = UNEXPLORED
    BFS(G,v)
```

O(m+n)

To find all v including not connected

detect cycle

Let G be a graph with n vertices and m edges. What is the worst case running time of Breadth First Search on G? Assume that the graph is represented using an **adjacency matrix**.

- (a) $O(n)$
- (b) $O(m + n)$
- (c) $O(m * n)$
- (d) $O(n^2)$

Consider two vertices x and y that are simultaneously on the queue during execution of BFS from vertex s in an undirected graph. Which of the following is/are true?

BFS

I. The number of edges on the shortest path from s to x is at most one more than the number of edges on the shortest path from s to y.

II. The difference in the number of edges from s to x and from s to y is at least 1.

III. There is a path from x to y.

- ○ (a) Only one of these statements is true.
- ◉ (b) Items I. and III. are true.
- ○ (c) Items II. and III. are true.
- ○ (d) Items I. and II. are true.
- ○ (e) All three statements are true.

Let G be a graph with n vertices and m edges. What is the worst case running time of Breadth First Search on G? Assume that the graph is represented using an **adjacency list**.

- ○ (a) $\Theta(n^2)$
- ○ (b) $\Theta(n)$
- ◉ (c) $\Theta(m+n)$
- ○ (d) $\Theta(m*n)$
- ○ (e) $\Theta(m)$

DFS

```
Algorithm DFS(G,v)
        Input: graph G and start vertex v
        Output:  labeling of the edges of G in the connected component of v as discovery edges and back edges

setLabel(v, VISITED)
For all w in G.adjacentVertices(v)
    if getLabel(w) = UNVISITED
      setLabel((v,w),DISCOVERY)
      DFS(G,w)
    else if getLabel((v,w)) = UNEXPLORED
      setLabel(e,BACK)
```

```
Algorithm DFS(G)
        Input: graph G
        Output:  labeling of the edges of G as discovery edges and back edges

For all u in G.vertices()
  setLabel(u, UNVISITED)
For all e in G.edges()
  setLabel(e, UNEXPLORED)
For all v in G.vertices()
  if getLabel(v) = UNVISITED
    DFS(G,v)
```

$\Theta(n+m)$

One version of DFS proceeds as: (1) Push the root with no associated edge. (2) While the stack is not empty, pop the next node and its associated edge (if any); if it has been marked as visited, skip it; otherwise, mark it as visited, mark its attached edge (if any) as a tree edge, and for each node adjacent to it, push the adjacent node along with the edge connecting to it from the current node.

DFS

Now, consider two vertices x and y that were simultaneously on the stack at some point during execution of DFS from vertex s in an undirected graph. Which of the following is/are true when DFS completes?

I. The number of edges on the shortest path from s to x is at most one more than the number of edges on the shortest path from s to y. X not deterministic

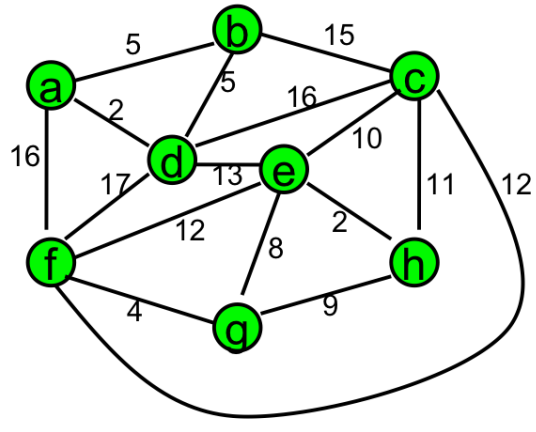
II. The difference in the number of discovery (tree) edges on the path from s to x and from s to y is at least 1. X

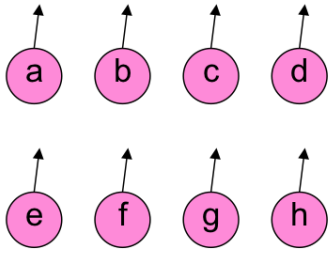
III. There is a path from x to y.

- (a) Item III. is true.
- ( ) (b) Item I. is true.
- ( ) (c) All three items are true.
- ( ) (d) None of the items is true.
- ( ) (e) Two of the items are true.

Kruskal's Algorism

Stop when n-1 edges added





(a,d)
(e,h)
(f,g)
(a,b)
(b,d)
(g,e)
(g,h)
(e,c)
(c,h)
(e,f)
(f,c)
(d,e)
(b,c)
(c,d)
(a,f)
(d,f)

1. Initialize graph T whose purpose is to be our output. Let it consist of all n vertices and no edges.

2. Initialize a disjoint sets structure where each vertex is represented by a set.

3. RemoveMin from PQ. If that edge connects 2 vertices from different sets, add the edge to T and take Union of the vertices' two sets, otherwise do nothing. Repeat until n-1 edges are added to T.

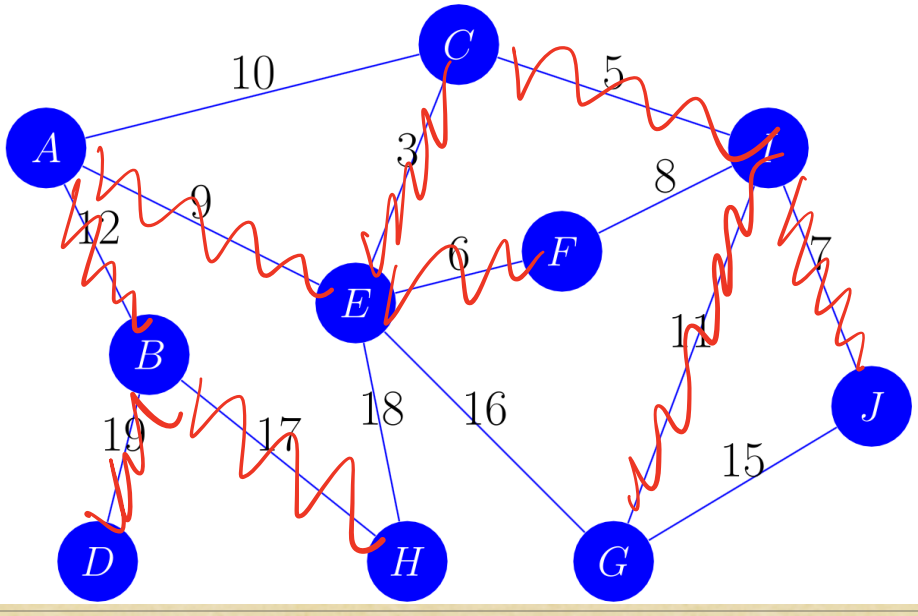
Priority Queue:	Heap	Sorted Array
To build	$\Theta(m)$	$\Theta(m \log n)$
Each removeMin	$\Theta(\log n)$	$\Theta(1)$

heap: $O(m \log n)$
array: $O(m \log n)$

Algorithm *KruskalMST*(*G*)

```
disjointSets forest;
for each vertex v in V do
    forest.makeSet(v);

priorityQueue Q;
Insert edges into Q, keyed by weights

graph T = (V,E) with E = ∅;

while T has fewer than n-1 edges do
    edge e = Q.removeMin()
    Let u, v be the endpoints of e
    if forest.find(v) ≠ forest.find(u) then
        Add edge e to E
        forest.smartUnion
            (forest.find(v),forest.find(u))

return T
```

Consider if we ran Kruskal's Algorithm on the following graph.



✓(C,E)
✓(C,I)
✓(E,F)
✓(I,J)
✗(F,I)
✓(A,E)
✗(A,C)
✓(G,I)
✓(A,B)
✗(G,J)
✗(E,G)
✓(B,H)
✗(E,H)

✓(B, D)

Suppose we implement Kruskal's algorithm using a heap as our priority queue and an adjacency matrix graph. What is the tightest worst case running time of the algorithm? (As usual, $|V| = n$, $|E| = m$, and you may assume that disjoint set functions run in constant time.)

- ○ (a) $O(n + m)$
- ○ (b) $O(m \log n)$
- ◉ (c) $O(n^2 + m \log n)$
- ○ (d) $O(n^2)$

Suppose we implement Kruskal's algorithm using an unsorted array as our priority queue and an adjacency list graph. What is the tightest worst case running time of the algorithm? (As usual, $|V| = n$, $|E| = m$, and you may assume that disjoint set functions run in constant time.)

- ○ (a) $O(n^2)$
- ◉ (b) $O(m^2)$
- ○ (c) $O(m \log n)$
- ○ (d) $O(n^2 + m \log n)$

Prim's Algorithm

Prim's Algorithm (undirected graph with unconstrained edge weights):

Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue
4. Initialize set of labeled vertices to ∅.

Repeat these steps n times:

- Remove minimum d[] unlabeled vertex: v
- Label vertex v (set a flag)
- For all unlabeled neighbors w of v,

  If cost(v,w) < d[w]

  d[w] = cost(v,w)

  p[w] = v

	adj mtx	adj list
heap	$O(n^2 + m \log n)$	$O(n \log n + m \log n)$
Unsorted array	$O(n^2)$	$O(n^2)$

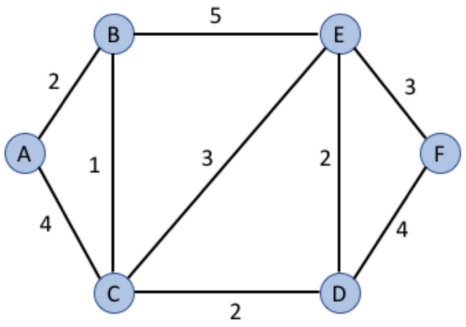
Which is best?

Depends on density of the graph:

Sparse $|E| = \Theta(n)$

Dense $|E| = \Theta(n^2)$

Prim's algorithm, when run on the following graph beginning at vertex A, produces a particular MST. What *other* vertex can we run Prim's on so that it produces a *different* MST?



- (a) C
- (b) F
- (c) D
- (d) None of the other options is correct.
- (e) E

Which of the following data structures should not be used to represent a dictionary?

- (a) Hash Table
- (b) Binary Search Tree
- (c) All of the other choices are reasonable dictionary implementations.
- (d) Heap
- (e) AVL Tree

Dijkstras Algorism

Initialize structure:

1. For all v, d[v] = "infinity", p[v] = null
2. Initialize source: d[s] = 0
3. Initialize priority (min) queue
4. Initialize set of labeled vertices to Ø.

Repeat these steps n times:

- Remove minimum d[] unlabeled vertex: v
- Label vertex v (set a flag)
- For all unlabeled neighbors w of v,

  If d[v] + cost(v,w) < d[w]

  d[w] = d[v] + cost(v,w)

  p[w] = v

	adj mtx	adj list
heap	$O(n^2 + m \log n)$	$O(n \log n + m \log n)$
Unsorted array	$O(n^2)$	$O(n^2)$

Which is best?

Depends on density of the graph:

Sparse

Dense

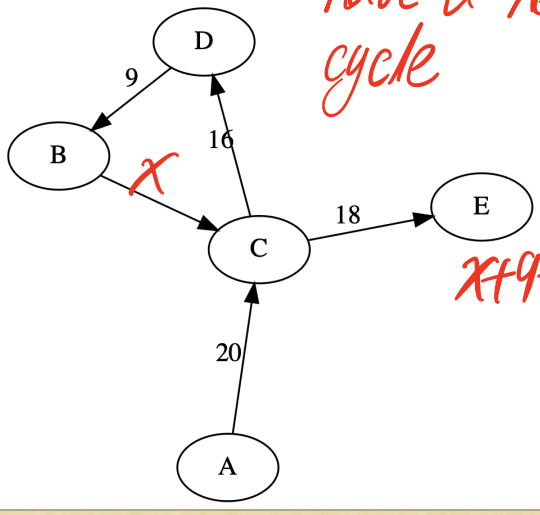
Which of the following can easily determine a **minimum** spanning tree?

- (a) DFS not minimum
- (b) Dijkstra's Algorithm not minimum
- (c) Prim's Algorithm
- (d) All of the other answers can easily determine a minimum spanning tree

For the graph below, state the largest integer weight for directed edge (B, C) that makes the given description true. (Note that the edge weight need not be positive!)

A shortest path from A to E does not exist.

have a net negative cycle



x+9+16<0

x<-25