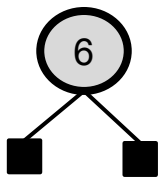
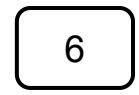
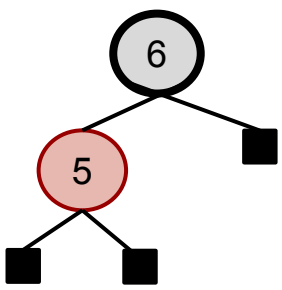
Solutions: Class 10/09, Balanced Trees

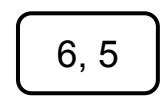


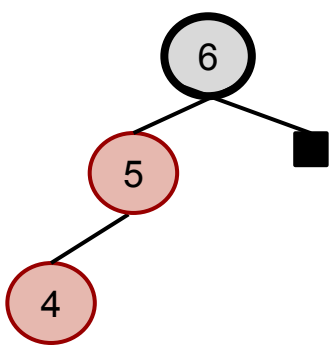
Insert 6:
← (a) RBT after insert
(b) **no** property is violated
(c) **no** remedy needed
(d) RBT after fixup is the same.
(e) (2,4): →



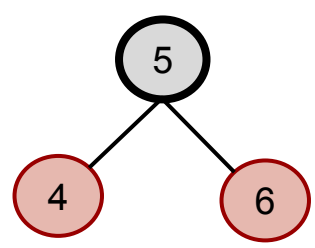


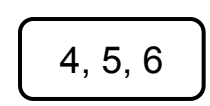
Insert 5:
← (a) RBT after insert
(b) **no** property violated
(c) **no** remedy needed
(d) RBT after fixup is the same.
(e) (2,4): →



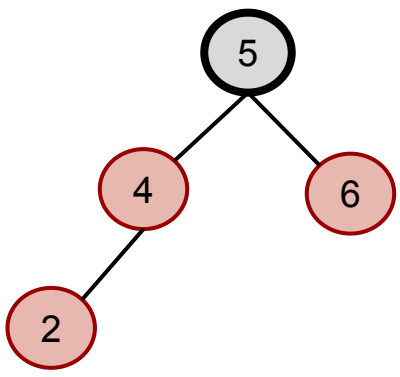


Insert 4:
← (a) RBT after insert
(b) **double red = incorrect representation**
(c) **black uncle** (sibling of red parent): **restructure**
(d,e) RBT and (2,4): →



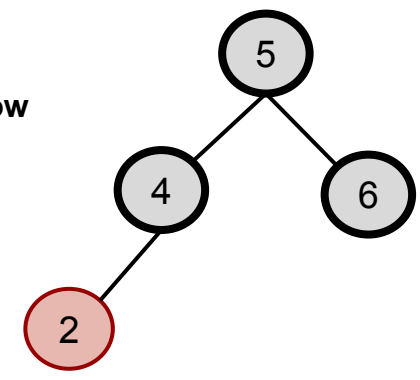


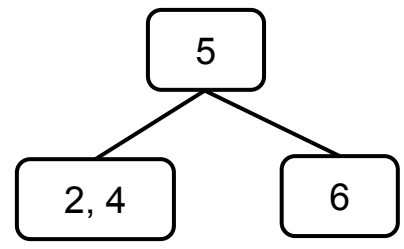
(You may leave out the black leaf nodes in the remainder. Show them only when they are the “uncle”)

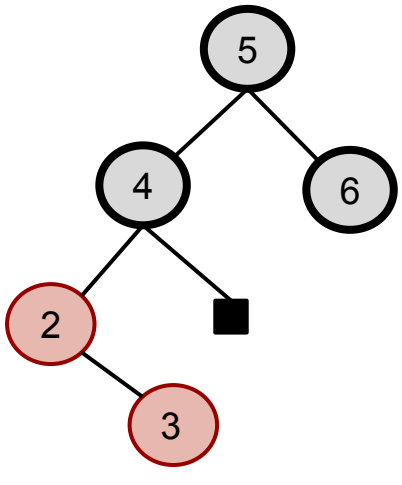


Insert 2:
← (a) RBT after insert
(b) **double red = overflow**
(c) **red uncle: recolor**
(d,e) RBT and (2,4): →

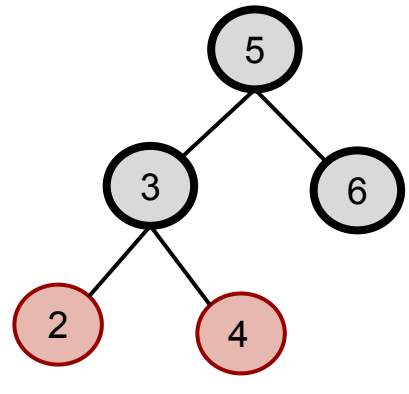
Note: 5 stays black because it is the root!

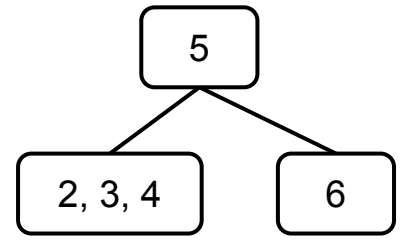




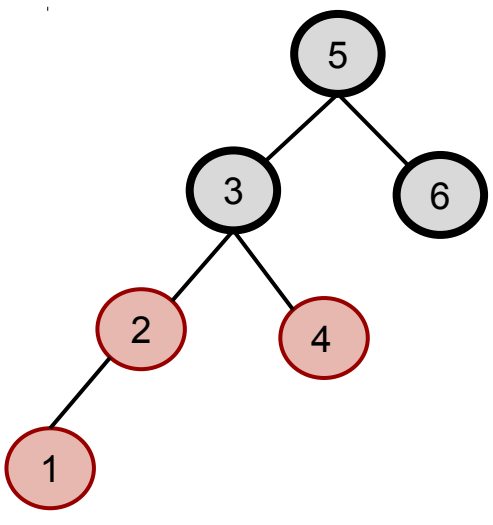


Insert 3:
← (a) RBT after insert
(b) **double red = incorrect representation**
(c) **black uncle: restructure**
(d,e) RBT and (2,4): →

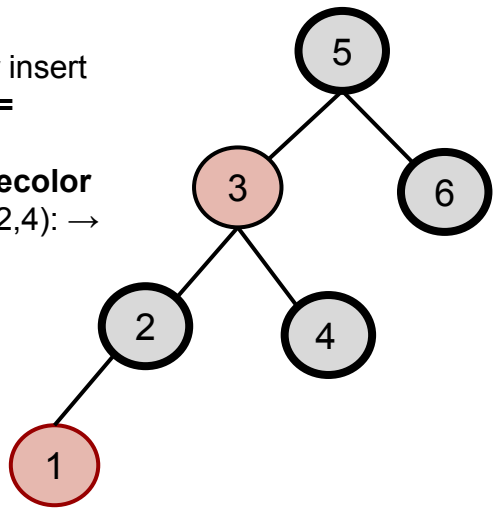


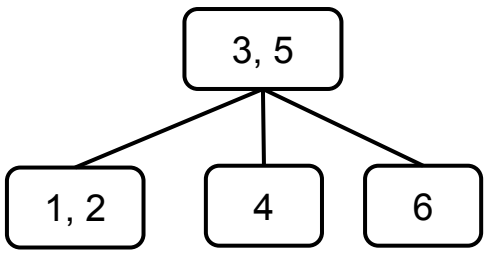


Solutions: Balanced Trees

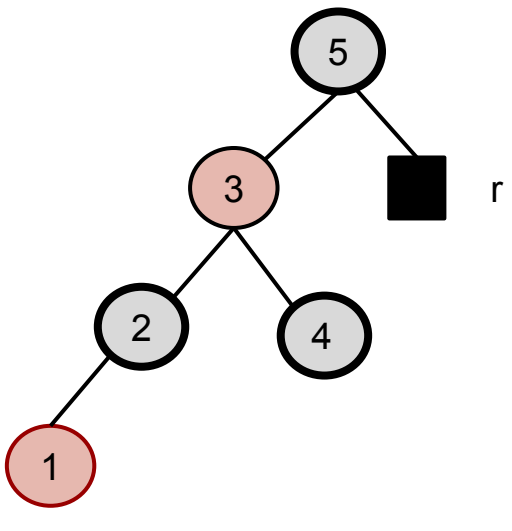


Insert 1:
← (a) RBT after insert
(b) **double red = overflow**
(c) **red uncle: recolor**
(d,e) RBT and (2,4): →

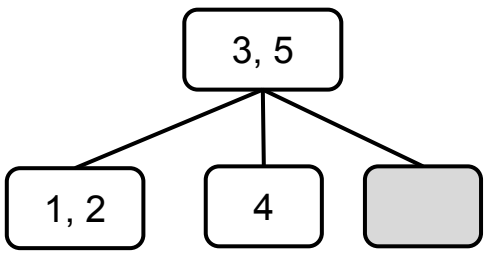




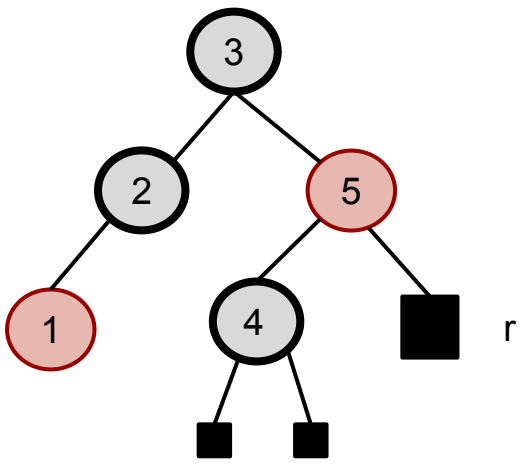
DELETION



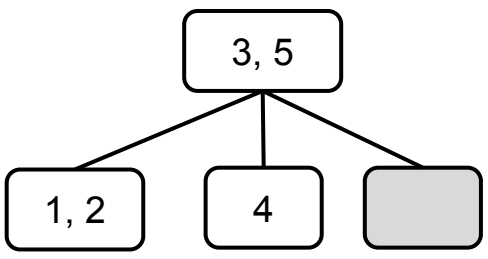
Delete 6:
(b): **double black at r**, corresponding to **underflow**
(c): **<u>Case 3</u>: sibling "3" is red: perform RBT <u>adjustment</u>.**
(Another case will then apply.)

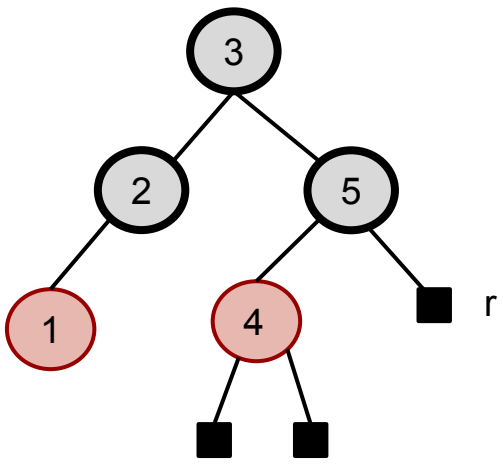


(Note: the text below departs from the template given students.)



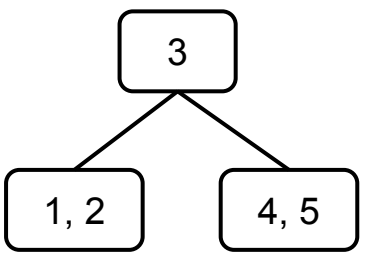
New Case:
(b): still **double black at r**, corresponding to **underflow,** but now ...
(c): **<u>Case 2</u>: sibling "4" is black with black children: perform RBT <u>recoloring</u>, equivalent to 2-4 fusion.**
(Essentially, push the blackness of both 4 and r up to parent 5.)

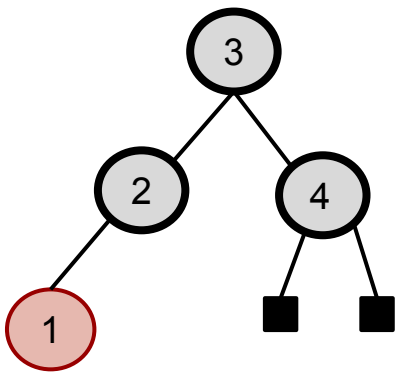




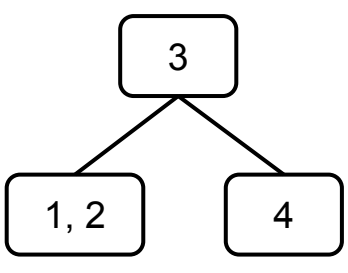
Now we are good. A red node absorbed the extra black by turning black.

(if 5 had been black the double black violation would propagate up the tree, potentially O(lg n) times.)





Delete 5:
BST deletion says when there is one child, replace it with its child.
RBT deletion says the child should take on its parent's color.
All is good.



Solutions -- Topic 9: Heaps

---

1. (1 point) Heap-Delete(A, i)

Procedure **Heap-Delete(*A*, *i*)** deletes the item at index *i* in heap *A* (represented as an array). **Give an implementation of Heap-Delete that runs in O(lg *n*) time** for a heap of size *n* = A.heapSize.

- Do not change the key of the item being deleted (in case it is an object referenced by another application).
- You may use instance variable A.heapSize.
- You do not need to include error checking.
- You do not need to return anything.
- *Hint: What other heap procedures do something similar? Use their code*

Comment: It is similar to Heap-Extract-Max, except that we are extracting max from a specified subtree. But since the replacement element may have come from a different subtree and therefore be larger, like Heap-Increase-Key we may need to propagate the replaced element up.

1 point. Half credit if there was some right idea but a logic error.

```
Heap-Delete(A,i)
// This part of solution comes from Heap-Extract-Max, and
// extracts the max from the subtree rooted at A[i]
1 A[i] = A[A.heapSize]
2 A.heapSize = A.heapSize - 1
3 Max-Heapify(A, i)
// Remainder of solution comes from Heap-Increase-Key, and
// ensures the new root of the subtree, which may have come
// from a different part of the heap, is smaller than its
// parents.
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

Comment: If we allowed you to change the key of the item being deleted, an

alternative solution follows. Advantage is the use of existing code and error checking, at small cost of more instructions executed. Disadvantage in an object oriented setting where heap entries are objects with accessor methods for keys is that the client application may not expect the key of the object to change.

```
Heap-Delete(A,i)
1 Heap-Increase-Key(A, i, ∞) // that is, infinity
2 Heap-Extract-Max(A)
```

2. (2 points) Heapsort on Sorted Data

Previously we noted that Insertion-Sort runs faster on already sorted data, but Merge-Sort does not. What about Heapsort? **How is it affected by sorted data? Give your reasoning: justification is more important than getting the answer right**. Refer to line numbers in code (appended) when discussing your analyses. *Hints: Consider Build-Max-Heap and the for loop separately. You may want to work examples, but don't get bogged down in details: return to asymptotic reasoning as soon as you see what is going on.*

For each: Give 0.5 points for saying O(n lg n) and 0.5 points for justification. For full credit, each part should discuss the effect of the increasing or decreasing order (only 0.25 points for justifications that do not do this).

(a) What is the **asymptotic running time of Heapsort** using a Max-Heap on an array *A* of *n* elements that is already sorted in ***increasing*** order? Justify your claim.

> **Solution:** O(n lg n). Build-Max-Heap is O(n) in general, and indeed it has to do some work to convert the sorted list into a max heap (which generally has the larger items earlier in the array). But the second part of Heapsort has a loop with O(n) passes, and each pass has a call to Max-Heapify at cost O(lg n), so the loop dominates with O(n lg n). If you work out detailed examples, you will see that once Build-Max-Heap is done, the array looks similar to a reverse-sorted array, since larger elements must be higher up in the tree = more towards the left. However, this does not help because Max-Heapify is called after putting one of the leaves in the root position; the leaf keys are small; and the one promoted has to propagate down the tree, making each pass O(lg n).

(b) What is the **asymptotic running time of Heapsort** using a Max-Heap on an array *A* of *n* elements that is already sorted in ***decreasing*** order? Justify your claim.

Solution: Also O(n lg n). Build-Max-Heap has to do less work in this case, as a reverse sorted list is already a max heap, so it does not have to actually swap anything (each call to Max-Heapify is constant time for the two assignments and conditional test). However, this is a constant reduction: Build-Max-Heap is still O(n) on reverse sorted data as it still runs the loop through half the items. Furthermore, the time savings does not matter: the loop that follows is O(n lg n), as it includes an O(lg n) Max-Heapify on each of the n items processed. The sorting does not reduce the O(lg n) passes, again because Max-Heapify is called after putting one of the leaves in the root position; the leaf keys are small; and the one promoted has to propagate down the tree.

3. (2 points) Ternary Heaps

You have just studied binary max-heaps. Suppose we want to use ternary max-heaps (each node has three children with smaller keys). Prove that a complete ternary heap of height h has **f(h)=(3^(h+1) - 1)/2** nodes.

Half a point for base case and 1.5 for remainder.

(a) Base case: $f(h=0) = (3^1 - 1)/2 = 1$

(b) Induction step: Assume formula is true for h. Prove $f(h+1) = (3^{h+2} - 1)/2$.

$f(h+1) = f(h) + 3^{h+1}$ // 3^{h+1} leaf nodes at level h+1

$= (3^{h+1} - 1)/2 + 3^{h+1}$ // substitute inductive hypothesis

$= (3^{h+1} - 1)/2 + (2 \cdot 3^{h+1})/2$

$= (3 \cdot 3^{h+1} - 1)/2$ // combine fractions

$= (3^{h+2} - 1)/2$ // QED

Topic 10A: Quicksort -- Solutions

Note: for simplicity we will use the nonrandomized version of Quicksort in the following problems. However, the results also apply to the randomized version.

1. Non-unique keys (warmup problem)

What is the running time of Quicksort (randomized or non-randomized) when all elements of the input array A have the same value? Justify your conclusion with reference to relevant lines of code.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Θ(n^2): Line 4 test of Partition includes equality, so will always succeed and execute lines 5 and 6. Everything in the partition will be placed to the left of (smaller side of) the pivot by lines 5 and 6. Thus the pivot position returned by line 8 will be the last element of the array.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Back in Quicksort, the recursion in line 3 will be on n-1 items (just removing the pivot) until the pivot is all that is left, and the recursion in line 4 will always be on 0 items (constant time each level of recursion).

Thus the sizes of the original call plus the recursive calls in line 3 will be on n + n-1 + n-2 + … + 1 + 0 = O(n^2) items, just as in the worst case analyzed in the lecture notes. In terms of recurrence relations, T(n) for a call to Quicksort is O(n) to partition in line 2, T(n-1) for recursion in line 3, and c or O(1) for recursion in line 4 (as well as the test of line 1). Putting it together and dropping the constant, the recurrence is T(n) = T(n-1) + O(n), which we have previously analyzed to be O(n^2)

2. Finding the *i*th smallest element with Partition

Use Quicksort's Partition procedure to write an algorithm for finding the i^{th} smallest element of an unsorted array in O(n) expected time (without sorting it). *Hint: what does the returned value of Partition tell you about the rank ordering of the pivot?*

- Describe the strategy in English
- Then write pseudocode for an algorithm.
- The solution is simpler if you assume arrays of A[1..n], so in the initial call p=1 and r=n. Solve the simpler version first.

- Then if you have time generalize for calls to any region of an array, e.g., the 5th largest element in A[37,1044].
- You do not need to prove O(n) expected time, but don't do something that takes longer such as sorting

a. Strategy described in English:

- Call partition on A(p,r) and get the returned pivot position q.
- If q = i and the original call was to A[1,n], we are done: return A[q]. The q^{th} position must be the i^{th} largest element because all smaller elements have been placed to the left of A[q] and there are exactly i-1 smaller elements.
- If q > i then the i^{th} largest element is in the lower half of the partitioned array: recursively find the ith smallest item in A[p,q-1].
- If q < i then the i^{th} largest element is in the upper half of the partitioned array, so we recursively find q = i in A[q+1,r]. This will work if the original call was to A[1,n], because we are assuming i is relative to the initial set of n elements, not the recursive set, and is therefore the same as the array index.
- However, if the lowest index was not originally 1, we need to adjust to convert between positional and absolute indexing, as well as to take into account the number of smaller elements discarded.

Conceptually, we are partially sorting the array only up to the point of knowing what would have ended up at the i^{th} position in a full sort.

b. Pseudocode, assuming initial call p=1:

```
Find-ith-element(A, p, r, i)
// Find i-th element of array A.
// we might insert error checking first: p <= i <= r?
1 q = PARTITION(A, p, r)
2 if q == i
3    return A[q]
4 else if q > i // search in lower partition
5    return Find-ith-element(A, p, q-1, i)
6 else          // search in upper partition
7    return Find-ith-element(A, q+1, r, i)
```

c. Pseudocode allowing any range of the array:

```
Find-ith-element(A, p, r, i)
// Find i-th element of subarray A[p,...,r]
1 q = PARTITION(A, p, r)
```

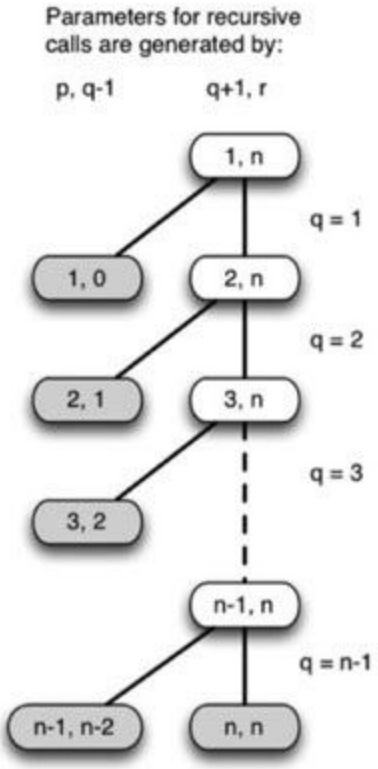
```
2 k = q - p + 1  // k is ordinal position of q in A[p,r]
2 if k == i      // is it the ith position?
3    return A[q] // then return the item at this index
4 else if k > i  // search in lower partition; i still holds
5    return Find-ith-element(A, p, q-1, i)
6 else           // adjust for discarded; search in upper
7    return Find-ith-element(A, q+1, r, i-k) // k were discarded
```

3. Challenge Problem: Calls to Partition in Worst and Best Case

We have seen that Quicksort requires $\Theta(n^2)$ comparisons in the worst case when the pivot is always chosen to be the smallest or largest element, and $\Theta(n \lg n)$ comparisons in the best case when the pivot is always the median key (or expected case for the randomized version). Here we examine the number of calls to Partition made in each of these cases.
In the following, do NOT assume that the number of calls to partition is the same as the overall runtime of quicksort. That is the point!

(a) Asymptotically, how many calls to Partition are made in the **worst case runtime** (when the pivot is always chosen to be the smallest or largest element)? Answer with Θ. Justify your conclusion, for example by using recurrence relations or reasoning about the recursion tree.

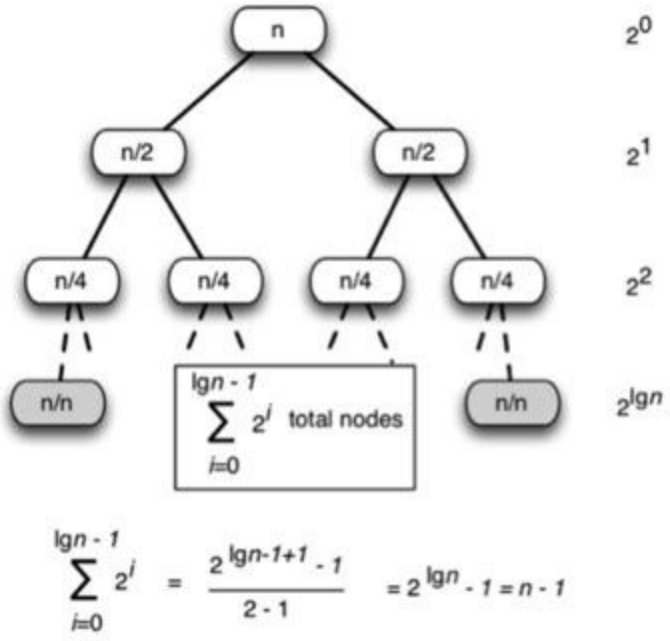
T(n) = T(n-1) + c = cn, so $\Theta(n)$ calls are made.



(b) Asymptotically, how many calls to Partition are made in the **best case runtime** (when the pivot is always the median key)? Answer with Θ. Justify your conclusion, for example by using recurrence relations or reasoning about the recursion tree.

$T(n) = T(n/2) + T((n/2)-1) + c \leq 2T(n/2) + c$
By the master theorem, a=2, b=2, $f=c=cn^0$.
$\log_2 2 = 1$ so subtract epsilon = 1 to get n^0: Case 1. $\Theta(n)$

The recursion tree:



$$\sum_{i=0}^{\lg n - 1} 2^i = \frac{2^{\lg n - 1 + 1} - 1}{2 - 1} = 2^{\lg n} - 1 = n - 1$$

A MUCH SIMPLER ARGUMENT: No matter how the recursion breaks down, it must proceed until every item has been 'picked off' as a pivot. So, n items must be picked off in n calls to Partition.

Solutions Topic 10b, 10/02: Linear and Stable Sorts

1. Stable Sort

(a) How can you modify any sorting algorithm to make it stable? Be specific about the additional data structures and changes to the algorithms needed.

> **Solution:** We need to record information about the original ordering, but we do not want to modify the original data. (Tricks like shifting the bits over and using lower order bits to record position will also work, but assume particular key data types.) A simple approach is to make the keys into (key, position) tuples and providing a comparison operator for these objects. The comparison operator will compare first on key, and then if they are equal compare on position to break the tie. This is modular, not requiring changing the sort code. A similar approach is to set up a parallel array of positions I[i] = i, and manipulate this array in parallel to the keyed elements: examples are given in the instructor presentation.

(b) How much additional time and space does your modification require? Justify your response.

> **Solution:** The additional space required is O(n lg n) because we record each of n positions and the largest position is n, which requires ⌈lg n bits⌉. The asymptotic time does not change because modification of the keys into tuples can be done in Θ(n) time and the extra comparison is constant time.
> **Note:** we have asked TAs to give full credit for Θ(n) space, as the point about the representation of n is less obvious.

2. Radix Sort

Use induction to prove that radix sort works. The base case is d=1. Induction will go from lower order to higher order digits. Your proof will need to rely on the assumption that the intermediate sort is stable. *(If it does not, your proof is incomplete! Hint: argue each case where the digits are <, =, and > separately.)*

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

> **Solution:**
> *Base case:* if d=1, sorting on that single digit sorts the array.
> *Induction:* Assume that radix sort works for d-1 digits. We need to show it works for d

digits.

(Radix sorts separately on each digit, starting with the low order digit 1. Radix sort of d digits is equivalent to sorting on the low-order d-1 digits followed by a sort on digit d. By induction the sort on the low order d-1 digits is correct, so the elements are in order according to these digits. The sort on the dth digit will order the elements by this digit. We need to show that this sort is correct for digits d..1)

Consider any two elements x and y with dth digits x_d and y_d.

- If $x_d < y_d$ the d^{th} pass of the sort will put x before y, which is correct, since x < y regardless of the low order digits.
- If $x_d > y_d$ the d^{th} pass of the sort will put y before x, which is correct, since y < x regardless of the low order digits.
- If $x_d = y_d$ the d^{th} pass of the sort will leave x and y in the same order they were in, because it is stable, But that order is already correct as it was determined by the sort on the d-1 digits.

Therefore the sort on the dth digit results in a correct overall sort.

Note: the above three cases do not imply that keys are compared. That would make it a comparison sort, and hence O(n lg n). The cases are for purposes of analysis only, to show that no matter what the relationship is on the d^{th} digit, the result will be correct.

Solutions -- Topic 12, Dynamic Programming

Longest Simple Path in a Directed Acyclic Graph

Given a **directed weighted acyclic graph G=(V,E)** and two vertices **s** (start) and **t** (target), develop a **dynamic programming approach** for finding a **longest weighted simple path** from **s** to **t**.

1. Characterize the Structure of an Optimal Solution

Let **p** be a **longest path from u to t**. If u = t then p is simply ⟨u⟩ and has zero weight. Consider when u ≠ t. Then p has at least two vertices and looks like: **p = ⟨u, v … t⟩** (it is possible that v = t).

Let **p' = ⟨v … t⟩** and **prove that p' must be a longest simple path from v to t.** (This is a proof of optimal substructure.)

> **Solution:**
> (Cut and paste argument:) Suppose that p' as defined above were *not* a longest simple path from v to t. Then there must exist some path p'' that is a longer simple path from v to t; that is (extending our w notation), w(p'') > w(p'). We can construct a new path p* from u to t consisting of u → v → p'' → t. This is a legal path because G is acyclic, so u cannot occur in p''. The length of this path p* is
>
> $$w(p^*) = w(u,v) + w(p'') > w(u,v) + w(p') = w(p)$$
>
> contradicting our definition of p as the longest simple path from u to t. Therefore, p' must be a longest simple path from v to t, and the problem exhibits optimal substructure.

2. Recursively define the value of an optimal solution:

Let **dist[u]** be the distance of a longest path from u to t. **Fill out the definition to reflect the above structure:**

$$\text{dist}[u] = \begin{cases} 0 & \text{if } u = t \\ \max_{v \in \text{Adj}[u]} \{w(u,v) + \text{dist}[v]\} & \text{if } u \neq t \end{cases}$$

3. Compute the value of an optimal solution (simple recursive version):

Write a recursive procedure that computes the value of an optimal solution as defined by the above recursive definition. Do not memoize yet; that's the next step.

Solution: (Notice how the code follows the mathematical definition.)

```
Longest-Path-Value-Recursive (G, u, t)
  if u = t
      return 0
  else
      max = -∞
      for each v in G.Adj[u]
          alt = w(u,v)
              + Longest-Path-Value-Recursive(G, v, t)
          if max < alt
              max = alt
      return max
```

4. Compute the value of an optimal solution (dynamic programming version):
Memoize your procedure by passing the array `dist[1..|V|]` that records longest path distances `dist[u]` from each vertex **u** to **t**. Assume that the **caller has initialized all entries of `dist` to -∞**.

Solution (additions highlighted):

```
Longest-Path-Value-Memoized (G, u, t, dist)
  if u = t
      dist[u] = 0  // caller may expect all entries defined
      return 0
  if dist[u] > -∞
      return dist[u] // this is the reuse of prior result
  else
      // dist[u] = -∞ is not necessary as caller did it
      for each v in G.Adj[u]
          alt = w(u,v)
              + Longest-Path-Value-Memoized(G, v, t, dist)
          if dist[u] < alt
              dist[u] = alt
      return dist[u]
```

5. Analyze the runtime of your solution in #4 in terms of |V| and |E|

Include

(a) the runtime to initialize **`dist`** and
(b) the runtime of **`Longest-Path-Value-Memoized`** itself.
(c) the resulting total runtime

This requires aggregating across loops.

(a) Θ(|V|) to fill in the entries.
(b) Θ(|E|) since in aggregate across all calls each edge is processed once(*), and all other operations are constant. (*) can be argued two ways: it is a DAG, so we never return to the same vertex, and even if we did we would just be doing constant lookup of paths we saved in dist.
(c) Total: Θ(|V| + |E|).

Solutions -- Topic 13, Greedy Algorithms

Activity Scheduling

In the activity scheduling problem, we are given the start and finish times s_i and f_i of a set S of candidate activities, and possibly other data about them. We know all the activities in advance: this is batch scheduling, not real time. We need to **select (schedule) a subset A of the activities that are *compatible* (do not overlap with each other)**. There are different versions of the problem depending on what we maximize.

Maximizing Count

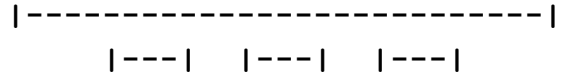
In this version of the problem we ***find the largest possible set (maximum <u>count</u>,*** not duration) *of activities that do not overlap with each other*. CLRS shows that this problem has the ***greedy choice property***: a globally optimal solution can be assembled from locally optimal choices. They show it by example with this *greedy strategy*: Always select the remaining compatible activity that **ends first**, and then solve the subproblem of scheduling the activities that start after this activity.

There are other greedy strategies, but some work and some don't. In the following two problems (and the extra credit problem) you determine whether alternative locally greedy strategies lead to globally optimal solutions. For each strategy below,

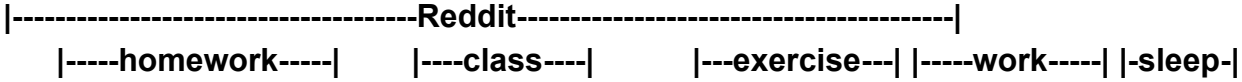
- **either show that the strategy leads to an optimal solution** by outlining the algorithm or approach (be sure to show it works on *any* input, not just the example you drew),
- **or give a counterexample**: write down a set of activities (defined by their start and finish times) that the strategy fails on, and show what goes wrong.

1. *Greedy strategy*: Always select the remaining compatible activity with the **earliest start time**. *Proposed Rationale:* Don't waste any time getting started.

Solution (1 pt): Counterexample:



Or as one group of students put it:



2. *Greedy strategy*: Always select the remaining compatible activity that has the **latest start time**. *Proposed Rationale:* Leave the most time remaining at the beginning for other activities.

Solution (2 pts, divided depending on approach taken, e.g., 1 for basic insight that it is the reverse form of the CLRS example and 1 for some demonstration that/how it works): This is the same as the greedy approach "ends first", but chronologically reversed in decreasing start times. Therefore we can modify the proof and the algorithm to work from the upper end of the activities' times sorted in decreasing order by start times, rather than from the lower end sorted in increasing order by finish times. Note: the inequality needs to be changed as well.

Algorithm recursive version: changes in yellow. We make a fictional activity a_0 with s[0] = ∞.

```
GreedyActivitySelect(s,f)
1   SortByDecreasingStartTimes(s,f)
2   n = s.length
3   return RecursiveActivitySelector(s,f,0,n)

RecursiveActivitySelector(s,f,k,n)
1   m = k + 1
// find the latest starting activity in Sk
2   while (m ≤ n) and (f[m] > s[k]) // finishes too late
3      m = m + 1
4   if m <= n
5      return { a_m } ∪ RecursiveActivitySelector(s,f,m,n)
6    else return ∅
```

Algorithm iterative version:

```
GreedyActivitySelect(s,f)
1   SortByDecreasingStartTimes(s,f)
2   n = s.length
3   A = { a_1 }
4   k = 1
5   for m = 2 to n
6      if f[m] ≤ s[k] // finishes on time
```

```
7           A ∪ { a_m }
8           k = m
9   return A
```

Different tables may take different approaches in attempting this: we'll give credit for showing they had the basic insight and tried to show it works.

Students are not required to do this, but a proof can be constructed following the proof in CLRS or Topic 13 by changing "earliest finish" to "latest start" and changing the comparison:

> ***Theorem:*** If S_k is nonempty and a_m has the ~~earliest finish~~ latest start time in S_k, then a_m is included in some optimal solution.
>
> *Proof:* Let A_k be an optimal solution to S_k, and let $a_j \in A_k$ have the ~~earliest finish~~ latest start time in A_k. If $a_j = a_m$ we are done. Otherwise, let $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ (substitute a_m for a_j).
>
> > *Claim:* Activities in A'_k are disjoint.
> >
> > *Proof of Claim:* Activities in A_k are disjoint because it was a solution.
> >
> > Since a_j is the ~~first~~ last activity in A_k to ~~finish~~ start, and ~~$f_m \leq f_j$~~ $s_m \geq s_j$ (a_m is the ~~earliest~~ latest in S_k), a_m cannot overlap with any other activities in A'_k
> >
> > No other changes were made to A_k, so A'_k must consist of disjoint activities.
>
> Since $|A'_k| = |A_k|$ we can conclude that A'_k is also an optimal solution to S_k, and it includes a_m. QED

Maximizing Value

We now consider a variation of the problem. Suppose that different activities earn different amounts of revenue. In addition to their start and finish times s_i and f_i, each activity a_i has revenue r_i, and our objective is now to **maximize the total revenue:**

$$\sum_{a_i \in A} r_i$$

3. Can you identify a greedy strategy that shows that this problem exhibits the greedy choice property? If so, show how it works to maximize value. If not, show counterexamples for the

strategies you considered and identify an alternative problem solving strategy that would work on this problem.

Solution (2 points: 1 for what won't work and 1 for what will): a greedy algorithm won't work. Counterexamples for the above strategies all apply (they all ignore the value). Counter examples also exist for value-optimizing methods:

Choose highest value first:

```
          |--100--|
|-------99-------|    |-------99-------|
```

Choose highest density of value per unit time first:
Above is also counterexample for this

This is similar to the 0-1 knapsack problem! Greedy won't work.
⇒ Use dynamic programming instead. (Students need not develop the solution.)

Challenge Problems: More Strategies for Maximizing Count

As you did in #1 and #2,

- either show that strategy below leads to an optimal solution
- or give a counterexample.

4. *Greedy strategy*: Always select the remaining compatible activity that has the **least duration**. *Proposed Rationale:* Leave the most time remaining for other activities.

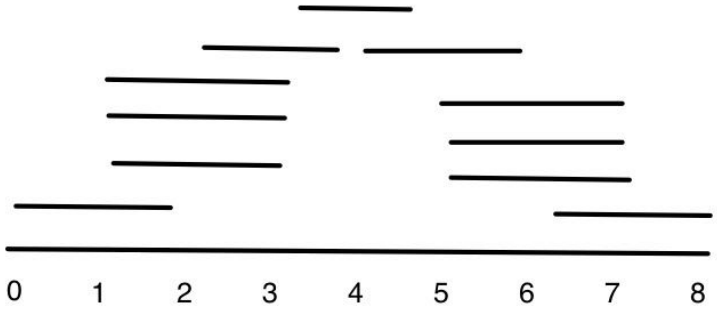
Counterexample (1 pt):

```
          |-------|
|----------------|    |----------------|
```

Perhaps that did not work because we don't care about duration, we care about how many other options are eliminated. So let's try …

5. *Greedy strategy*: Always select the remaining compatible activity that **overlaps with the fewest number of remaining activities**. *Proposed Rationale:* Eliminate the fewest number of remaining activities from consideration.

Counterexample (1 pt):



Middle activity will be chosen first, forcing choice of only one activity on each side. However, it is possible to get a sequence of 4 activities.

Solutions, Topic 14A: Graph Representations

Efficient Graph Transposition

The transpose of a directed graph G = (V, E) is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u)$ such that $(u, v) \in E\}$. In other words, G^T is the graph G with all of its edges reversed.

Algorithms for computing transpose by *copying* to a new graph are trivial to write under matrix and adjacency list representations. However, we will be working with very large graphs and want to avoid copies. Can we transpose a graph "in place" without making a copy, and minimizing the amount of extra space needed. What is the time and space cost to do so?

(In the following, you are writing algorithms that work "inside" the ADT: you are allowed to access the matrix and list representations directly. Examples of representations are on the next page.)

1. (1pt) Describe an efficient algorithm for the **edge list** representation of G for computing G^T from G in place (don't make a copy). Do this by modifying the list. Analyze the space and time requirements of your algorithm.

> **Solution:** Since an edge-list representation is a sequence of edges $\langle$ ... (i, j) ... $\rangle$, we can access all the edges sequentially in O(E) time. For each edge, we use a temporary variable to swap i and j, producing (j, i). This takes O(1) space. This is the simplest and fastest way to compute a transpose, but edge-lists are not a good representation for many of the other operations we need to do.

2. (2 pts) Describe an efficient algorithm for the **adjacency matrix** representation of G for computing G^T from G in place (don't make a copy). Do this by modifying the matrix. Analyze the space and time requirements of your algorithm.

> **Solution:**
> Flipping the bits (0 to 1 and 1 to 0) will not work: this makes the complement graph. The solution is to swap entries A[i,j] and A[j,i] for every pair of (i,j). We must be careful not to do it twice for each pair (therefore, leaving the matrix unchanged). To achieve that we just need to iterate for every element below the main diagonal of the matrix and swap each element with the corresponding element above the main diagonal:

```
for i = 2 to n {
     for j = 1 to i-1 {
```

```
            temp = A[i,j];
            A[i,j] = A[j,i];
            A[j,i] = temp;
      }
}
```

Note that j is always less than i, so if we have processed this with A[k,l] as the cell assigned to temp we will never inadvertently reverse it with A[l,k] as the cell assigned to temp. Also, no cell A[i,i] is processed as that would be pointless. The loops can also be written with i = 1 to n-1 and j = i+1 to n (*What does that correspond to in our high level description of the solution?*).

Runtime is $O((V^2-V)/2) = O(V^2)$ time, and requires O(1) additional space (just one temp variable).

3. (2 pts) Describe an efficient algorithm for computing G^T from G using the **adjacency list** representation of G, using as little extra space as possible. Can you do it "in place"? Analyze the space and time requirements of your algorithm.

Solutions: Adjacency lists are space efficient for large sparse graphs, but they achieve this at the cost of requiring sequential access to the vertices adjacent to a given edge. One can't go directly to a data item saying whether vertices x and y are adjacent: one must search for y on the adjacency list for x.

All of the solutions we are aware of require nontrivial additional space, but we can minimize this space. Let's consider some alternatives, starting with the naive solution:

A. Copying the graph while reversing edges.
This solution is the default solution we are trying to avoid: making an entirely new graph instance. Make a new vertex array, and then iterate over the vertex array of the original graph and over the adjacency list for each vertex to copy the edges in reverse. Specifically if a linked list cell referencing vertex y is found on the adjacency list for vertex x, we make a linked list cell referencing vertex x and place it at the front of the adjacency list for vertex y. This is conceptually simple, but assuming we also keep the original graph requires O(V+E) additional space and O(V+E) time. For a sparse graph, O(V+E) is close to O(V), while for a dense graph it is closer to $O(V^2)$.

If we did not want to keep the original graph, we might delete the old edges from the original graph as they are copied to the new one so that they can be garbage collected as we proceed: we create O(E) new edge objects, but we also free up edge objects at

the same rate, so their memory can be reclaimed if needed. However, garbage collection takes time that is difficult to analyze without details of the programming language implementation. Yet this gives us the idea of managing the reuse of memory ourselves:

B. Copying while reusing linked list cells.
Use plan A, copy the graph, but as we copy each edge to the new graph, delete the linked list cell for y (found on x's list) from the original graph and re-use that cell when creating a linked list cell for x (to be put on y's list). Then we will never need any more than O(V) additional storage for the second vertex array. When done, we can either return a new graph, or make each vertex in the original graph point to the new adjacency lists and dispense with the second vertex array in order to modify the original graph in place. Time cost is the same as copying.

Other solutions have been proposed to avoid making the second O(V) vertex array:

C. Using existing adjacency lists with marked list cells.
The basic idea is the same as B in which we reuse the O(E) list cells, but instead of constructing new lists off a second O(V) array, we construct those lists within the original linked lists using the original O(V) array. However, here we run into a problem similar to the one we saw in problem 2 with matrix representations: We need a way to keep track of which linked list cells represent original edges and which represent edges that have been reversed, so that if we encounter a linked list node for vertex y on x's adjacency list and put a linked list node for x on y's adjacency list, we don't later move it back when processing y's adjacency list. (Note that unlike with the matrix this error would not necessarily restore the graph to its original state: y may or may not actually be processed after x.)

One way to do this is to "mark" linked list cells. We could add an extra bit to each linked list cell, set to 0 in the original and 1 when reversed. Then, when traversing an adjacency list we only reverse (remove and reuse on a new list) those that are set to 0, and then set the bit to 1. Students have also proposed using a hash table to keep track of reversed edges (what would be the key?) But both of these solutions require O(E) extra space. O(E) can range from O(V) in sparse graphs to $O(V^2)$ in dense graphs, so any list cell marking approach is potentially worse than plan B above, depending on the graph.

D. Tail pointers to separate existing and new adjacency lists.
There is another approach to keeping track of reversed edges that requires only O(V) additional space: a tail pointer for each adjacency list. This solution was first proposed to us by students. This solution assumes that you have a tail pointer to the end of each

original linked list. (It would cost O(E) time to set them up if they are not already present.) The tail pointers will mark the boundaries between the old and new lists, serving simultaneously as the sentinel for the end of the old list and the head of the new list. Similarly to Plan C, for each reversal, remove the node from the list you found it in (preserving the remaining list), and add the node for the reversed edge on the appropriate list. Different from Plan C, add the nde *after the node the tail pointer points to*. When processing a linked list to reverse its target nodes, *stop when you reach the same cell as the tail pointer*: the remaining linked list will be the reversed edges. (At this point, the tail pointer is the head pointer for the new list, and we no longer have a tail pointer for the new list unless we also maintained that during the above process.)

Of all the solutions presented here, Plan B (a second O(V) vertex array) and Plan D (O(V) tail pointers) are the most space efficient solutions for adjacency lists. Plan D requires additional code for careful pointer manipulation, while Plan B uses standard list manipulation methods. I would recommend Plan B as the code will be easier to write and maintain, but appreciate the creativity of students in thinking of Plan D.

4. (Challenge Problem). Now describe a way to compute G^T from G under the **adjacency matrix** representation without modifying the matrix, and by using only one bit of extra storage for the entire graph!!! Analyze space and time requirements. *(Hint: take advantage of the ADT abstraction layer.)*

Solution: The one extra bit will indicate whether the graph is transposed or not.

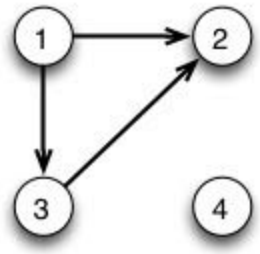
Initially the bit is set to 0. The transposeGraph() operation sets it to the complement of the current value (i.e. 1 if it was 0, and 0 if it was 1). Then we need to modify all the other Graph methods so that they check the bit and if it is set to 1, they access the graph as if it had been transposed, by reversing the source and target as appropriate. For example, a call to getArc(i,j) would actually get arc (j,i); inAdjacentVertices(v) would return out-adjacent vertices and outAdjacentVertices(v) would return in-adjacent vertices; Origin and Destination would be flipped, and similarly for the mutators, taking care to add or remove arcs in the reverse direction. When transposeGraph() is called again, the bit is set to 0 and the normal versions of the methods are used.

This requires one bit of extra storage for the graph which is O(1). It is O(1) to transpose and the run times of the other methods are not affected beyond constant factor overhead to check the bit value.

Given this solution, why would one want to modify the graph as in question 2? The code becomes more complex to maintain, so it is worth it only if transpose is a common operation. Why not use the same approach with adjacency lists? Without

random access to (i, j) the "flipped" operations would become much more expensive.

Examples:



Formal Specification

G = (V, E)
V = {1, 2, 3, 4}
E = {(1, 2), (1, 3), (3, 2)}

Matrix Representation

(Rows and columns are indexed by vertex number.)

	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0
4	0	0	0	0

Fastest to test whether (i, j) in E, but slower to get all adjacent vertices. Not space efficient for sparse graphs.

Adjacency List Representation

Left hand column is an array indexed by vertices and containing pointers to linked lists. The order of the lists is *not* significant: they are representing sets.

```
1 → v2 → v3 → null
2 → null
3 → v2 → null
4 → null
```

Fastest to get a list of adjacent vertices; time to test whether (i, j) in E is proportional to average degree, usually OK in sparse graphs. Space efficient.

Edge List Representation

```
(1, 2)
(1, 3)
(3, 2)
```

Edges are listed in no particular order, and are accessed sequentially. Fastest to iterate over all edges but very costly to find all the edges out of a vertex. No way to represent unconnected vertices or any vertex attributes: vertices are not first class objects. Space efficient.

Topic 14B Depth First Search Solutions

(corrected 11/14)

1. *(0.5)* Using one color to find discovery and finish times in a directed graph. The DFS-VISIT procedure as written uses three colors; WHITE, GRAY, and BLACK. Show that we can find discovery and finish times with two colors, WHITE and one other, and hence only one bit of storage per vertex. Do this by **modifying the relevant code, and arguing that the algorithm still assigns discovery and finish times correctly.** (Note: three colors are needed for some other applications.)

Instruction added in class: Please mark **changes with boldface** and ~~deletions with Format/Text/Strikethorough~~.

Modified Code (edit this):

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```

Why it works:
The only tests of color are in line 6 of DFS and line 5 of DFS-Visit. These tests only care whether or not the vertex is white. In the existing algorithm, Gray and Black will lead to the same outcome in these tests. Therefore we need only one color.

The non-white color indicates that a node has just been discovered, so should not be processed again in lines 6-7 of DFS-Visit. Therefore we should mark the node non-white as soon as it is visited, in line 3. (If line 3 were not there, one might get back

to u via a cycle and try to process it again.) It does not matter whether we call it “gray” or “black, as long as the color becomes non-white in line 3. Having done this, line 8 serves no purpose and can be deleted.

---

2. *(1.5)* Finding connected components of an undirected graph.
Modify the relevant code to find the connected components of an **<u>undirected</u>** graph G (these are NOT SCC). Specifically, **assign to each vertex v an integer label v.cc taking on values from 1..k where k is the number of connected components.** Thus, v.cc == u.cc only if v and u are in the same connected component.

Post-class Comment: The above instructions will be modified to clarify that v.cc is the component ID. We’ll probably call it v.CID. We want to use consecutive IDs 1, 2, … k.

Modified Code (edit this):

```
DFS-Find-Components(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  component = 0
6  for each vertex u ∈ G.V
7      if u.color == WHITE
8          component++
9          DFS-VISIT-FC(G, u)
```

```
DFS-VISIT-FC(G, u)
1  time = time + 1
2  u.cc = component
3  u.d = time
4  u.color = GRAY
5  for each v ∈ G.Adj[u]
6      if v.color == WHITE
7          v.π = u
8          DFS-VISIT-FC(G, v)
9  u.color = BLACK
10 time = time + 1
11 u.f = time
```

Why it works:
Each call to DFS-VISIT-FC finds all vertices reachable from u. Since one can go over edges in either direction in an undirected graph, the vertices reached are by definition in the same connected component, so we should label them all with the same component ID. The changes to DFS-Find-Components ensure that each call to DFS-VISIT-FC is under a unique ID.

This solution leaves in the bookkeeping for parent π, discovery time and finishing time, but they could be deleted if we were not interested in those results. (See next solution.)

3. *(3)* Detecting cycles in directed and undirected graphs.

Some graph algorithms (e.g., Topological Sort) won't work on graphs with cycles. Therefore it is useful to determine whether a graph has a cycle. We don't have to find all cycles: just return TRUE if the graph has one and FALSE if not. Modify the code below to do this: the same code should work on both directed and undirected graphs. Then justify why your solution leads to the correct result and analyze time complexity for the two types of graphs.

Modified Code: (2pts)

```
  DFS-Has-Cycle(G)
  1  for each vertex u ∈ G.V
  2       u.color = WHITE
  3       u.π = NIL
  4  time = 0
  5  for each vertex u ∈ G.V
  6       if u.color == WHITE
  7            if DFS-VISIT-HC(G,
u)
  8                 return TRUE
  9  return FALSE
```

```
  DFS-VISIT-HC(G, u)
  1  time = time + 1
  2  u.d = time
  3  u.color = GRAY
  4  for each v ∈ G.Adj[u]
  5       if v.color == GRAY &&
             v != u.π
  6            return TRUE
  7       if v.color == WHITE
  8            v.π = u
  9            DFS-VISIT-HC(G, v)
 10  u.color = BLACK
 11  time = time + 1
 12  u.f = time
 13  return FALSE
```

Why it works:

We need to find the "back edges": edges that go back to a vertex that is still being actively explored. If a graph has a cycle, then the DFS search that starts from one vertex of the cycle will eventually follow a back-edge to that vertex. Conversely, if a DFS search finds a back-edge, there must be a cycle involving the vertex reached by that back-edge. This is where we need more than two colors: these will be the vertices that have been colored GRAY in line 3 of DFS-Visit, but are encountered again in a recursive call (line 10) before they are finished by lines 8-10 and colored BLACK.

With undirected graphs, there is an ambiguity (discussed in CLRS in the section on Classification of Edges): If we go over the edge {u, v} in the direction (u, v) and then in the recursive call go over it in the direction (v, u), is this a back edge? No: each edge must receive one classification, and it is already classified as a tree edge. The problem

is that our code annotates vertices, not edges, so we need to check whether we are going back to the immediate parent of u to exclude this situation. Thus the lines for π are restored.

So, as soon as we see GRAY for a vertex v that is not the parent of u we report TRUE. The rest of the code changes are just to get this report back out the top level (treating DFS-VISIT-HC as a logical predicate), and to return FALSE if it is never reported.

Note: a previous edition of these solutions had an error in that it did not check for the parent situation just discussed. This prior edition stated that the parent π, discovery, and finishing time bookkeeping is not needed and should be deleted because the early exit caused by finding a cycle will leave those values incomplete for some vertices. We now know that we need the parent π. It would be best to implement this annotation in a manner not affecting the client's view of the graph.

Asymptotic time complexity on directed graphs (0.5pt):
$O(V + E)$. This is the same as for the original DFS on directed graphs, because it is possible we will have to explore all vertices and edges. (See below for an example.)

Asymptotic time complexity on undirected graphs (0.5pt):
One might think that this is also O(V + E), but in an undirected graph with |V| vertices you cannot have more than |V| − 1 edges without having a cycle, since a tree has |E| = |V| − 1 edges and a cycle is formed if you add one more edge. So, it is impossible to see |V| edges without having detected a cycle along the way: it is O(V).

The reason that this argument does not apply to directed graphs is it is possible to have a DAG with more than |V| − 1 edges, including cases where the edge set grows faster than |V|. For example, in a DAG of n vertices where E contains (v_i, v_j) for all j > i, the edge set size |E| = n(n-1)/2 = $O(V^2)$. A small example for n=4: G = ({a, b, c, d}, {(a, b), (a, c), (a, d), (b, c), (b, d), (c, d)}}.

Topics 15-16: Amortized Analysis & Union-Find Solutions

A space-efficient strategy for table growth

When we allocate a new table of twice the size and copy the old table into the new one, the new table is only half full. Prof. Efficiency thinks this is a waste of space and suggests the following solution: instead of doubling the size of the table when it becomes full, increase the table size by a fraction ε (epsilon) instead and copy the elements into this new table. For example, when ε = .5, if a table is full and contains *m* elements, the newly allocated table has capacity $\lceil 1.5m \rceil$. This way, we are using only at most 50% more space than actually is needed to store the elements in the table. Here we will analyze the effect of Prof. Efficiency's proposal on the time to insert into such a table.

1. Parameterizing the Algorithm (0.5pt). The code for Table-Insert from the lecture notes is below, with ε added as a parameter `epsilon`. Change the code to use `epsilon`, **boldfacing** your changes. Remember that table size is an integer.

```
Table-Insert (T,x,epsilon)
  1  if T.size == 0
  2      allocate T.table with 1 slot
  3      T.size = 1
  4  if T.num == T.size
  5      allocate newTable with ceiling((1+epsilon)*T.size) slots
  6      insert all items in T.table into newTable
  7      free T.table
  8      T.table = newTable
  9      T.size = ceiling((1+epsilon)*T.size)
 10  insert x into T.table
 11  T.num = T.num + 1
```

2. Deriving the Amortized Time per Operation. Let's derive a general formula for the time to insert into a table, given ε. Assume we increase the size of the table by an arbitrary constant epsilon ε. Assume the table T contains *m* elements and is full. (Use *m* in your formulas, not T.size.)

(a, 0.5 pt) Total Capacity: What will be the size (total capacity) of the new the table T' after resizing? (This will also be the number of items that have to be copied when T' is resized)

Answer: $(1+\varepsilon)m$ … or ceiling($(1+\varepsilon)m$))

(b, 0.5 pt) Available Space: How many elements can we insert into T' before it gets resized?

Answer: εm … or ceiling(εm)

(c, 0.5 pt) Credit Available for Next Resize: Assume each insertion into the table costs us $1, but we charge customers $$c$ > $1 for each insertion in Part (b) in order to raise credit for covering the cost of resizing the table when it gets full. How much credit will we have accumulated by the time T' has to be resized? Give your answer in terms of c, m, and ε.

Answer: We spend one CyberDollar to do the insertion, and the rest (c-1) are the credit. Therefore, we have $(c-1)\varepsilon m$ credit.

(d, 1.0 pt) Amortized Cost of Each Insertion: How much should we charge per each insertion to make sure we have enough credit to copy all elements next time we resize the table? That is, what is *c* as a function of ε? Using the expressions you wrote above, set up an equation where credit ≥ cost and solve for c.

Answer: To copy all elements at next resizing, we need to copy $(1+\varepsilon)m$ items. Thus, to have enough credit, we need to solve the equation

$$(c-1)\varepsilon m \geq (1+\varepsilon)m$$

which solves to

$c \geq 1 + \frac{1+\varepsilon}{\varepsilon}$ or 2 + 1/ε (or equivalent)

3. Specific cases. Now let's evaluate the amortized time-cost of Prof. Efficiency's proposal for specific ε. Compute the cost of insertion if:

(a, 0.25 pt) ε = 0.5 (Prof. Efficiency's proposal.)

Answer: $2 + \frac{1}{0.5} = 4$

(b, 0.25 pt) ε = 0.1 (Prof. Stingy thinks that Prof. Efficiency has not gone far enough.)

Answer: $2 + \frac{1}{0.1} = 12$

(c 0.25 pt) ε = 1.0 (Doubling of the table, as in the CLRS version.)

Answer: $2 + \frac{1}{1} = 3$

(d, 0.25 pt) ε = 4.0 (Prof. Generous has plenty of space to spare.)

Answer: $2 + \frac{1}{4} = 2.25$

(e, 1 pt) Conclusions: What do the above results tell us about time/space tradeoff in tables with expansion? How does this tradeoff affect the big-O amortized cost of such tables?

- Allocating more space makes the amortized cost per operation faster (and less makes it slower) - 0.5 pt
- but the amortized cost is still O(1) - 0.5 pt.

This is another classic time/space tradeoff, although there are diminishing returns for large ε since it is in the denominator.

---

4. Challenge Problem: Iterative Find-Set

The Find-Set method for Dynamic Disjoint Sets as written in CLRS Section 21.3 is not tail recursive, so a compiler won't be able to automatically generate efficient iterative executable code. Write an iterative version of Find-Set that uses path compression.

Of course, many variations are possible, but they must (a) do path compression and then (b) return the root of the tree in the end. Since you don't know who the root is until you reach it, vertices passed along the way must be saved. The recursive solution does this implicitly with the runtime stack, suggesting that we use an explicit stack to record vertices on the path, and then pop the stack to make them point to the root once it is found.

```
Find-Set-Iterative(x) {
    S = new Stack()
    while x != x.p {
        S.push(x)
        x = x.p }
    // now x is at the root of the tree
    // pop all vertices visited and do path compression
    while !S.isEmpty() {
        S.pop().p = x }
   return x
}
```

But one can also do it without a stack, just by climbing the parent pointers twice. This is more efficient: why allocate a stack object when you just need the path?

```
Find-Set(x)
  r = x           // we need the x
  while r != r.p
    r = r.p
  // now r is the root of the tree
  // set all parents on path to r to do path compression
  while x != r
    p = x.p       // preserve the parent
    x.p = r       // compress
    x = p         // go to parent
  return x        // or r
```

A common error is to forget to set the parent of *all* vertices on the path to x.

Topic 17, Minimum Spanning Trees: Class Solutions

Kruskal's and Prims on Unconnected Graphs

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

```
PRIM(G, w, r)
1  Q = ∅
2  for each u ∈ G.V
3      u.key = ∞
4      u.π = NIL
5      INSERT(Q, u)
6  DECREASE-KEY(Q, r, 0)        // r.key = 0
7  while Q ≠ ∅
8      u = EXTRACT-MIN(Q)
9      for each v ∈ G.Adj[u]
10         if v ∈ Q and w(u, v) < v.key
11             v.π = u
12             DECREASE-KEY(Q, v, w(u, v))
```

1. Suppose we have an <u>unconnected</u> graph, and want to find a minimum spanning <u>forest</u> (consisting of a minimum spanning tree for each connected component).

a. Will Kruskal's algorithm correctly find a minimum spanning forest in an unconnected graph? Why or why not?

> **Solution:** Yes. It considers all edges in the graph, so all edges in each connected component will be considered, thereby applying the algorithm to each component.

b. Will Prim's algorithm correctly find a minimum spanning forest in an unconnected graph? Why or why not?

> **Solution:** Yes. At first one might think that Prim's cannot reach components not connected to the start vertex *r*, but examining the operation of the loop more carefully, vertices in other components will eventually be dequeued (with key of ∞) in line 7, and construction of the MST for the given component will commence from the first such vertex dequeued. Hence, choice of r as a starting point is truly arbitrary: we could have left all keys at ∞.

Proposed MST Algorithms and their Implementations

As a working computer scientist, colleagues may propose or you may think of new algorithms to solve a problem. Critically evaluating such proposals is an important skill. Below

are three proposed MST algorithms that operate on an undirected graph G = (V, E). (One is a challenge problem.) For each algorithm,

- either prove that the set of edges *T* produced by the algorithm is always a minimum spanning tree for any G,
- or find a counter-example G where *T* is not a minimum spanning tree (either because it is not a tree, or it is not of minimum weight).
- Then, regardless of whether the algorithm is correct, identify an efficient implementation of the utility methods used. (Consider modifying existing algorithms.)

2. Maybe-MST-A

```
Maybe-MST-A (G, w)
1  T = {} // empty set
2  for each edge e ∈ E, taken in arbitrary order
3      if T ∪ {e} has no cycles
4          T = T ∪ {e}
5  return T
```

a. Does this construct MSTs? Give a proof or counter-example:

Solution: Maybe-MST-A is not a valid MST algorithm. Counter-example:
Maybe-MST-B simply constructs a tree, with no attempt to guarantee minimality. A simple counter-example is a triangle: if the two highest cost edges are chosen first, a higher cost tree will be constructed.
Notice the similarity to the Connected-Components algorithm, which also examines edges in arbitrary order to add those that do not form cycles.

b. Describe an efficient algorithm for detecting cycles after adding an edge e, as in:

```
3      if T ∪ {e} has no cycles
```

Solution: Depth-first search in O(*V*+*E*) time, exiting as soon as a back edge is found. Possible efficiency gain if you start the search with one of the vertices on the edge just added, as we know any cycle must involve this vertex, but this is still O(*V*+*E*). *Note:* this is DFS-Has-Cycle, which you wrote last week!

3. Maybe-MST-B

```
Maybe-MST-B (G, w)
1  sort the edges into nonincreasing order of edge weights w
2  T = G.E
3  for each edge e, taken in sorted order
4      if T – {e} is a connected graph
5          T = T – {e}
6  return T
```

a. Does this construct MSTs? Give a proof or counter-example:

Solution: Maybe-MST-A is a correct MST algorithm.
Note that Maybe-MST-A always maintains a connected graph, i.e. in the end, the remaining edges will constitute a spanning tree because it is connected and we have deleted the maximum number of edges we can without disconnecting it. We just need to prove that this spanning tree is minimum weight.

To prove by contradiction, suppose that when the algorithm terminates, T contains an edge e that connects two components c_1 and c_2 and could be replaced with a lower cost edge e' that must also connect these components. But then due to the ordering in line 3, e would have been processed and removed earlier than the lower weight e', because c_1 and c_2 would have remained connected by e', contradicting the assumption that the algorithm could terminate with e in T.

b. Describe an efficient algorithm for testing connectivity, as in:

```
4       if T – {e} is a connected graph
```

Solution: Depth-first search in $\Theta(V+E)$ time, checking that all vertices are reached on the first pass.

Kruskal avoids this work because it *constructs* rather than deconstructs the tree: it can then use efficient union/find, with amortized cost $O(\alpha(V))$. We can't reverse this for removing edges, as the union-find operations don't record what edge(s) led to two vertices being in the same set. This is why the constructive rather than destructive approach is preferred.