CSE 373: Data Structures and Algorithms

Lecture 17: Finish Dijkstra's Algorithm, Preserving Abstractions (Software Design), Spanning Trees

Instructor: Lilian de Greef
Quarter: Summer 2017

Dijkstra's Algorithm (Pseudocode)

Dijkstra's Algorithm – the following algorithm for finding all the single-source shortest paths from one particular source vertex, in a weighted graph (directed or undirected) with no negative-weight edges:

1. For each node `v`, set `v.cost = ∞` and `v.known = false`
2. Set `source.cost = 0`
3. While there are unknown nodes in the graph
   a) Select the unknown node `v` with lowest cost
   b) Mark `v` as known
   c) For each edge `(v,u)` with weight `w`,

```
c1 = v.cost + w   // cost of best path through v to u
c2 = u.cost       // cost of best path to u previously known
if(c1 < c2){      // if the path through v is better
  u.cost = c1
  u.path = v      // for computing actual paths
}
```

Correctness: Intuition

Rough intuition:

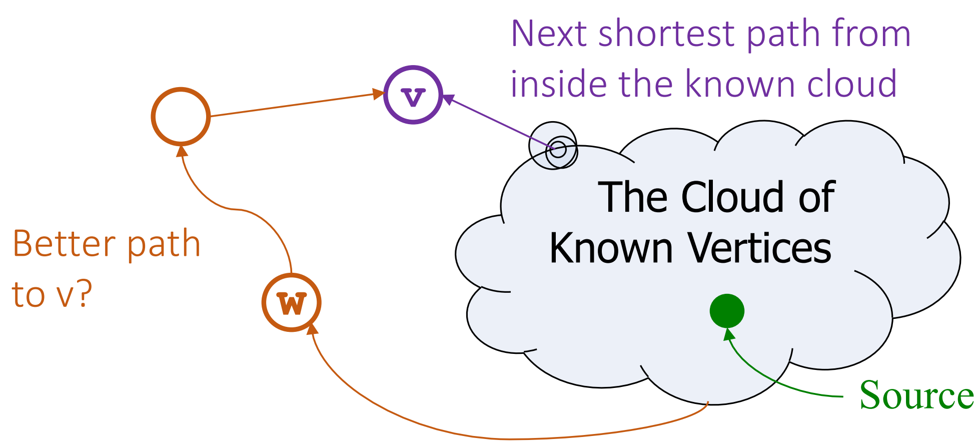
All the "known" vertices have the correct shortest path

- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node "known", then by induction this holds and eventually everything is "known"

Key fact we need: When we mark a vertex "known" we won't discover a shorter path later!

- This holds only because Dijkstra's algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction…

Correctness: The Cloud (Rough Sketch)



- Suppose **v** is the next node to be marked known (next to add to "the cloud of known vertices")
- The best-known path to **v** must have only nodes "in the cloud"
  - Else we would have picked a node closer to the cloud than v
- Suppose the actual shortest path to **v** is different
  - It won't use only cloud nodes, or we would know about it
  - So it must use non-cloud nodes. Let w be the *first* non-cloud node on this path.
  - The part of the path up to w is already known and must be shorter than the best-known path to v.
  - So v would not have been picked. Contradiction!

Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
       if(b.cost + weight((b,a)) < a.cost){
         a.cost = b.cost + weight((b,a))
         a.path = b
       }
}
```

Improving asymptotic running time

- So far: $O(|V|^2)$
- We had a similar "problem" with topological sort being $O(|V|^2)$ due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges
- Solution?
  - A holding all unknown nodes,
  - But must support operation
    - Must maintain a reference from each node to its current position in the priority queue
    - Conceptually simple, but can be a pain to code up

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
     if(!a.known)
      if(b.cost + weight((b,a)) < a.cost){
        decreaseKey(a,"new cost – old cost")
        a.path = b
      }
}
```

Dense vs. Sparse (again!)

- First approach: $O(|V|^2)$
- Second approach: $O(|V|\log|V|+|E|\log|V|)$
- So which is better?
  - Dense or Sparse? $O(|V|\log|V|+|E|\log|V|)$ (if $|E| > |V|$, then it's $O(|E|\log|V|)$)
  - Dense or Sparse? $O(|V|^2)$
- But, remember these are worst-case and asymptotic
  - Priority queue might have slightly worse constant factors
  - On the other hand, for "normal graphs", we might call `decreaseKey` rarely (or not percolate far), making $|E|\log|V|$ more like $|E|$

Preserving Abstractions

A software-design interlude from Graphs

Memory “under the hood”: Stack Space and Heap Space

Code

```
int x;
int x = 2;
int y = x;
y = 4;
return x;
```

```
Date today = new Date(2017,7,31)
Date tomorrow = today;
tomorrow.addDate();
return today.getMonth();
```

```
Class Date {
  int year;
  int month;
  int day;
}
```

COMPUTER MEMORY

Stack Space

Heap Space

(extra space for notes / scratch work)

Abstractions

The key idea of code **abstraction**:

- Clients do not know how it is implemented
- Clients do not *need* to know
- Clients cannot “break the abstraction” *no matter what they do*

Abstraction: Separation of Clients and Implementation

Data Structure Client:

"not trusted by ADT implementer"

- Can perform any sequence of ADT operations
- Can do anything type-checker allows on any accessible objects

```
Priority Queue
Example:

new PQ(…)

insert(…)

deleteMin(…)

isEmpty()
```

Data Structure Code:

- Should document how operations can be used and what is checked (raising appropriate exceptions)
- If used correctly, correct priority queue for any client in this example
- Client "cannot see" the implementation
  - e.g. binary min heap

Our example

- A priority queue with to-do items, so earlier dates "come first"
- Exact method names and behavior not essential to example

```
public class Date {
   … // some private fields (year, month, day)
   public int getYear() {…}
   public void setYear(int y) {…}
   … // more methods
}
public class ToDoItem {
   … // some private fields (date, description)
   public void setDate(Date d) {…}
   public void setDescription(String d) {…}
   … // more methods
}
public class ToDoPQ {
   … // some private fields (array, size, …)
   public ToDoPQ() {…}
   void insert(ToDoItem t) {…}
   ToDoItem deleteMin() {…}
   boolean isEmpty() {…}
}
```

What's the mistake?

```
public class ToDoPQ {
   … // other fields
   public ToDoItem[] heap;
   public ToDoPQ() {…}
   void insert(ToDoItem t) {…}
   …
}
// client:
pq = new ToDoPQ();
pq.heap = null;
pq.insert(…); // What will likely happen here?
```

Today's lecture: `private` does not solve all your problems!

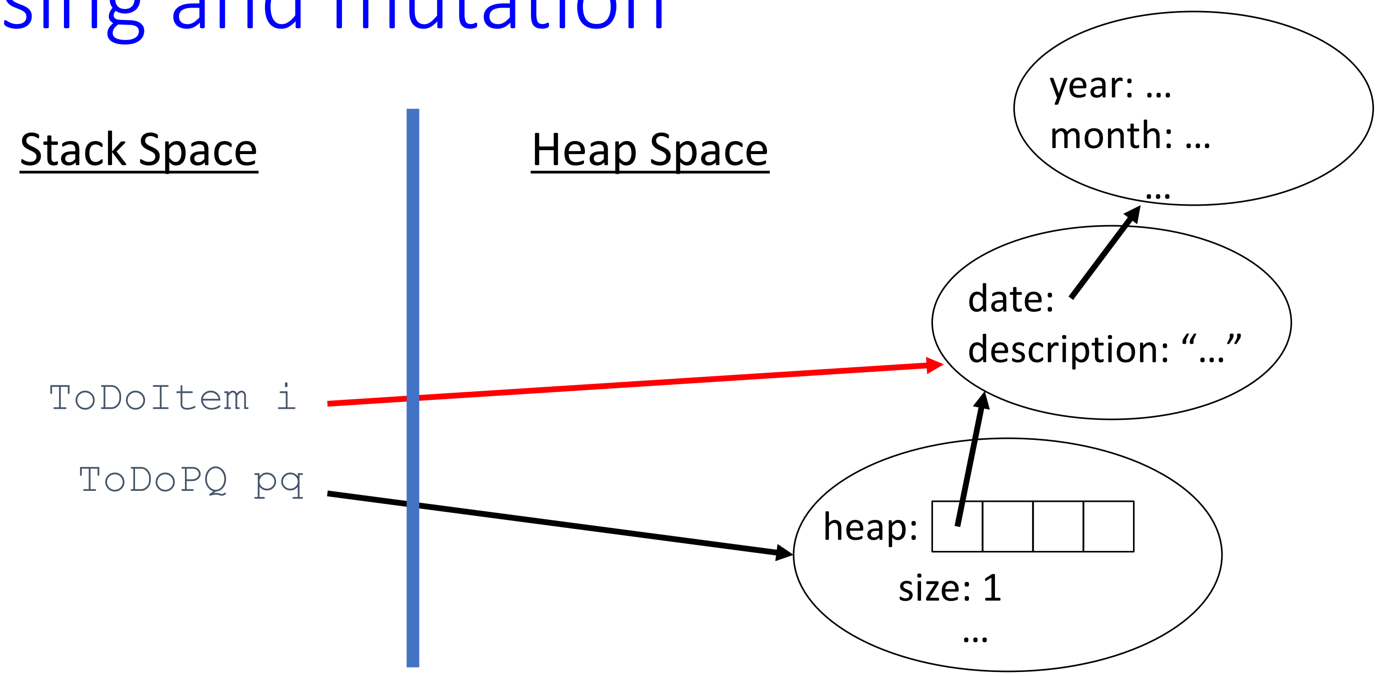
Upcoming pitfalls can occur even with all `private` fields

Less obvious mistakes

```
public class ToDoPQ {
   … // all private fields
   public ToDoPQ() {…}
   void insert(ToDoItem i) {…}
   …
}

// client:
ToDoPQ pq = new ToDoPQ();
// Make item with description “do a thing”
ToDoItem i = new ToDoItem(…);
pq.insert(i);
i.setDescription(“eat pie”);
pq.insert(i); // same object after update
x = deleteMin(); // x’s description???
y = deleteMin(); // y’s description???
```

Aliasing and mutation



- Client was able to update something inside the abstraction because client had an alias to it!
- It is too hard to reason about and document what should happen, so better software designs avoid the issue

Practice:
What year does x have? What happens on the last line?

```
ToDoPQ   pq = new ToDoPQ();
ToDoItem i1 = new ToDoItem(…); // year 2013
ToDoItem i2 = new ToDoItem(…); // year 2014
pq.insert(i1);
pq.insert(i2);
i1.setDate(…); // year 2015
x = deleteMin(); // What year does x have?

ToDoItem i3 = new ToDoItem(…);
pq.insert(i3); // year 2016
i3.setDate(null);
ToDoItem i4 = new ToDoItem(…); // year 2017
pq.insert(i4); // What happens here?
```

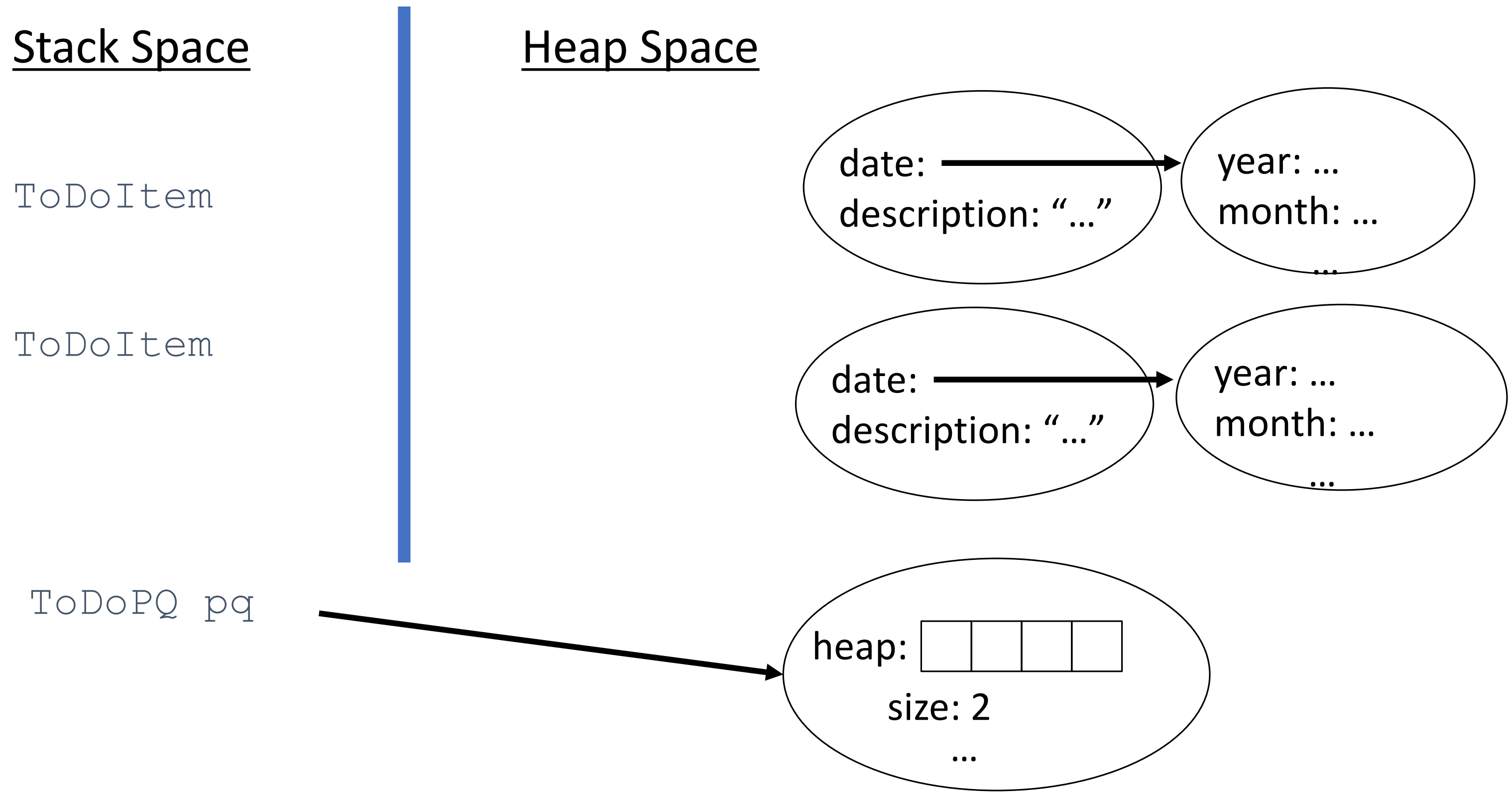
A) 2014,
inserts item for 2017.

B) 2015,
inserts item for 2017.

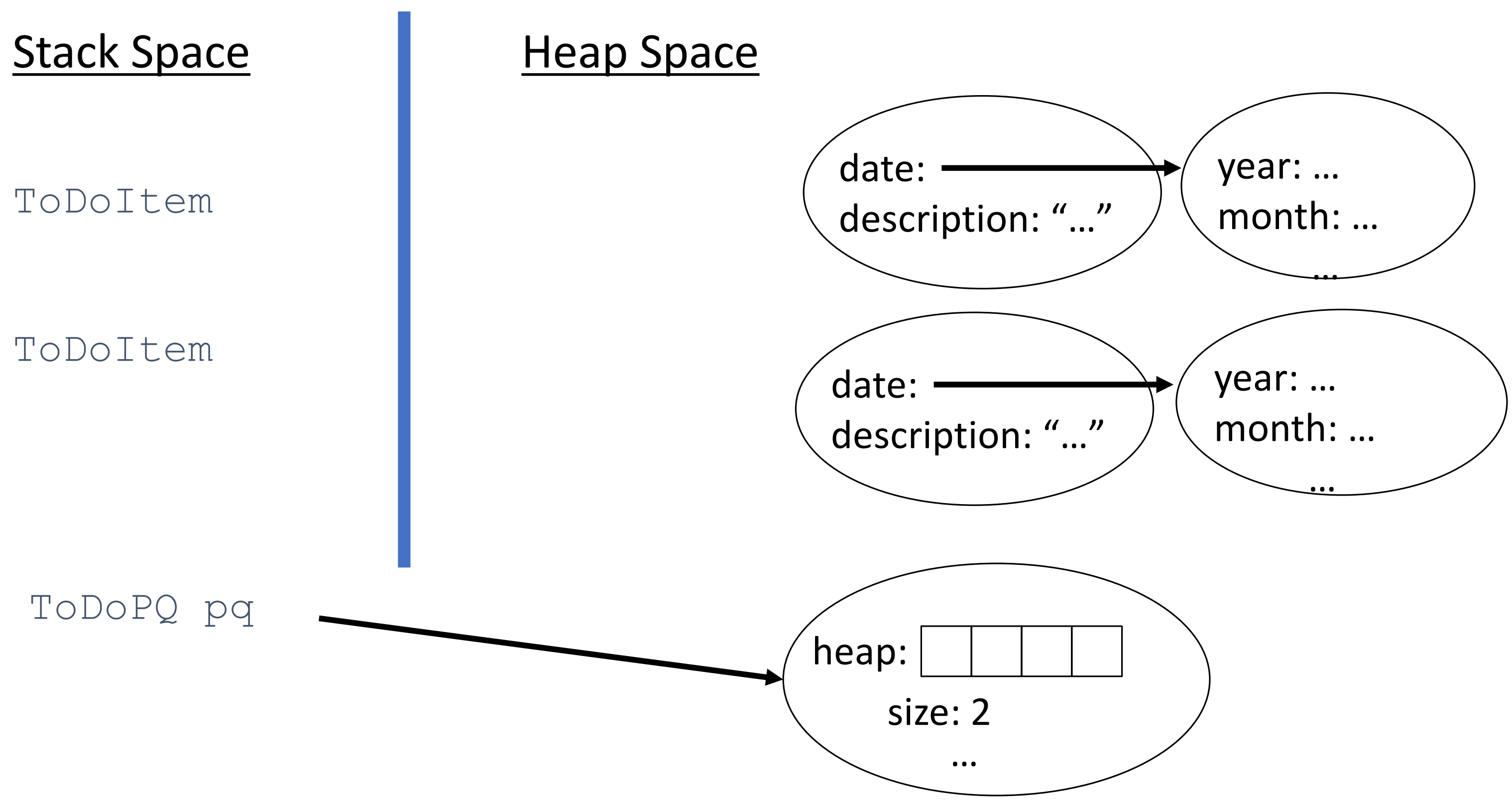
C) 2014,
throws exception.

D) 2015,
throws exception.

Practice


Stack Space
Heap Space
ToDoItem
ToDoItem
ToDoPQ pq
date:
description: "…"
year: …
month: …
…
date:
description: "…"
year: …
month: …
…
heap:
size: 2
…

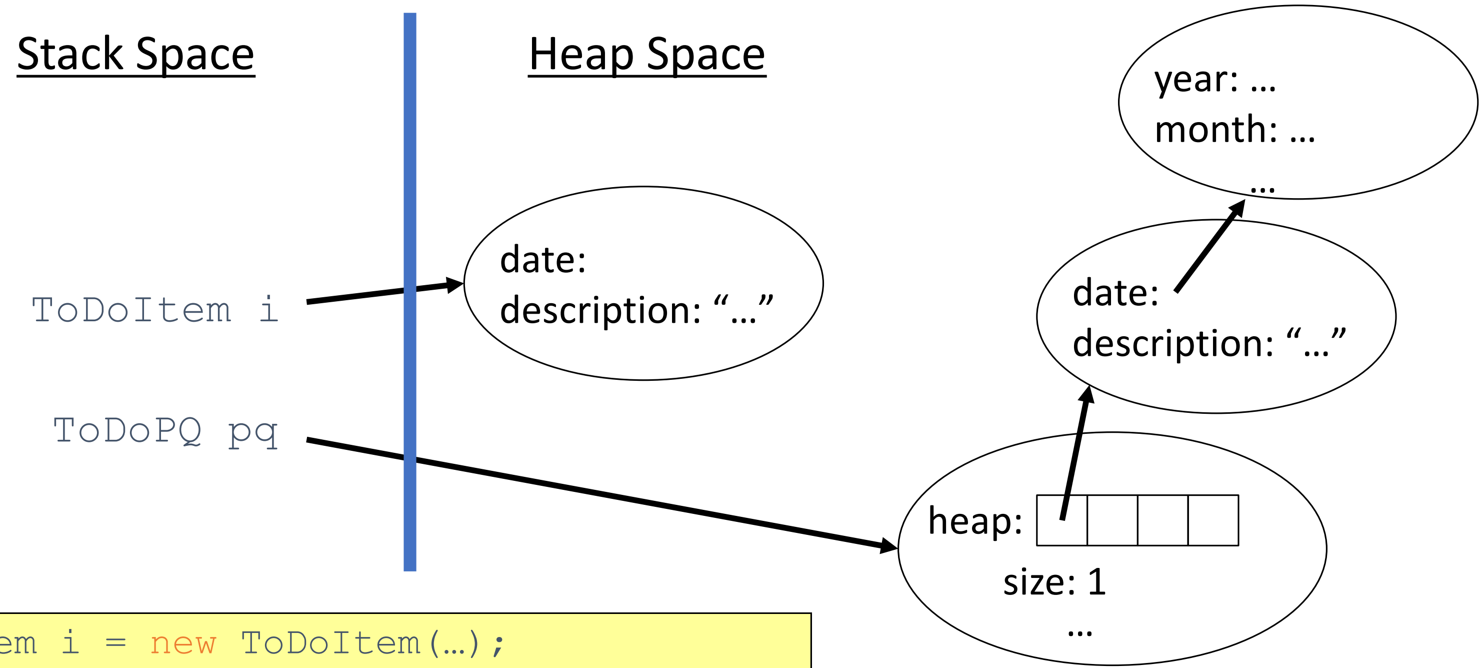
Practice


Stack Space
Heap Space
ToDoItem
ToDoItem
ToDoPQ pq
date:
description: "…"
year: …
month: …
…
date:
description: "…"
year: …
month: …
…
heap:
size: 2
…

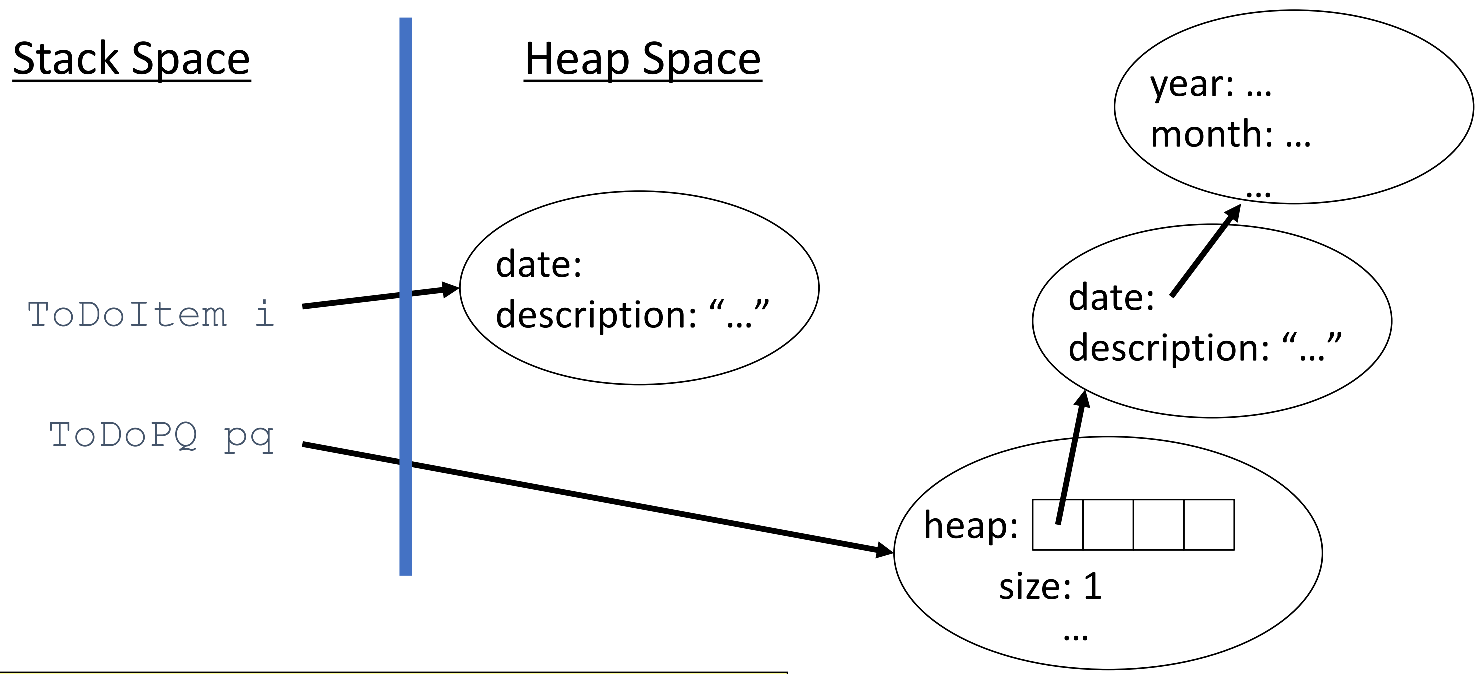
The general fix

- Avoid aliases into the internal data (the "red arrows") by copying objects as needed
  - Do not use the same objects inside and outside the abstraction because two sides do not know all mutation (field-setting) that might occur
  - **"Copy-in-copy-out"**
- A first attempt:

```
public class ToDoPQ {
   …
   void insert(ToDoItem i) {
     ToDoItem internal_i =
       new ToDoItem(i.date, i.description);
     … // use only the internal object
   }
}
```




```
ToDoItem i = new ToDoItem(…);
pq = new ToDoPQ();
pq.insert(i);
i.setDescription("some different thing");
pq.insert(i);
x = deleteMin();
y = deleteMin();
```




```
Date d = new Date(…)
ToDoItem i = new ToDoItem(d, "buy cake");
pq = new ToDoPQ();
pq.insert(i);
d.setYear(2015);
…
```

Deep copying

- For copying to work fully, usually need to also make copies of all objects referred to (and that they refer to and so on…)
  - All the way down to `int`, `double`, `String`, …
  - Called **deep copying** (versus our first attempt *shallow-copy*)

- Rule of thumb: Deep copy of things passed into abstraction

```
public class ToDoPQ {
   …
   void insert(ToDoItem i) {
     ToDoItem internal_i =
       new ToDoItem(new Date(…),
                    i.description);
     … // use only the internal object
   }
}
```

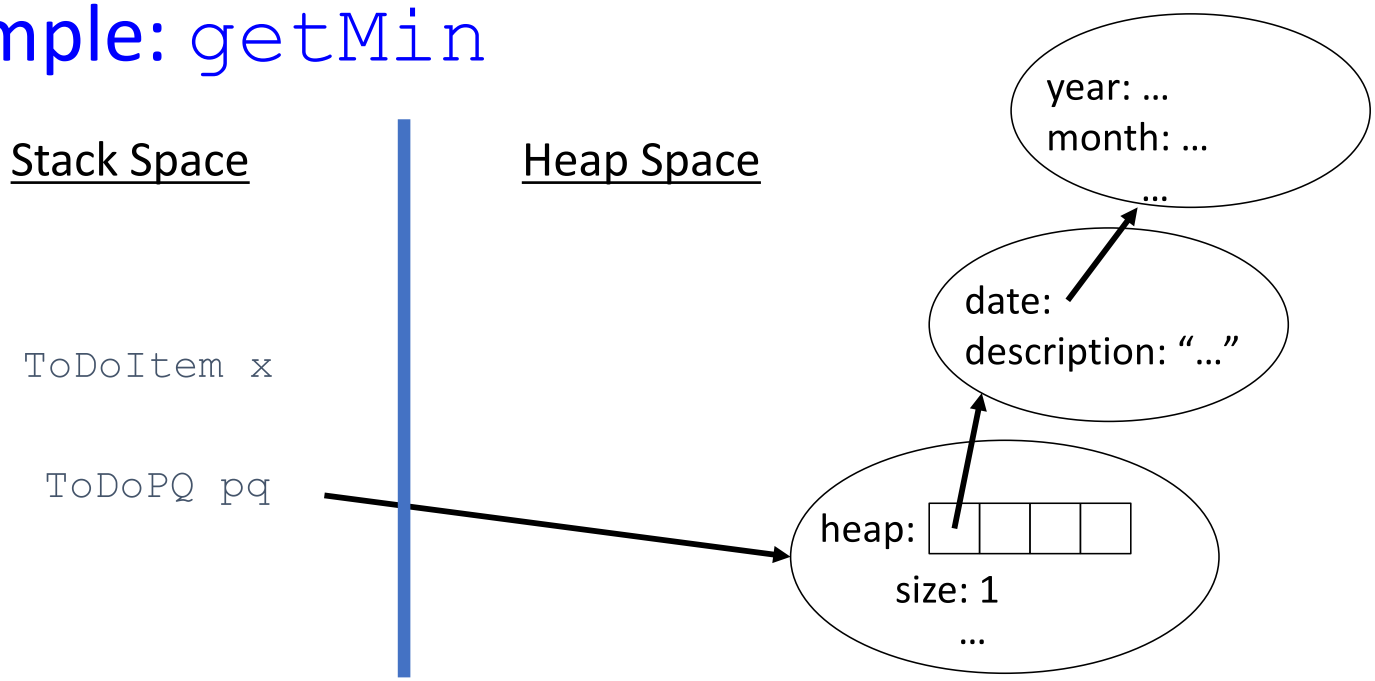
That was copy-in, now copy-out…

- So we have seen:
  - Need to deep-copy data passed into abstractions to avoid pain and suffering

- Next:
  - Need to deep-copy data passed out of abstractions to avoid pain and suffering (unless data is “new” or no longer used in abstraction)

- Then:
  - If objects are immutable (no way to update fields or things they refer to), then copying unnecessary

Example: `getMin`




```
ToDoItem i = new ToDoItem(…);
pq = new ToDoPQ();
ToDoItem x = pq.getMin();
x.setDate(…); // Uh oh!
```

```
public class ToDoPQ {
  ToDoItem getMin() {
  ToDoItem ans = heap[0];
    return ans;
  }
}
```

The fix: Copy-Out

- Just like we deep-copy objects from clients before adding to our data structure, we should deep-copy parts of our data structure and return the copies to clients

- Copy-in *and* copy-out

```
public class ToDoPQ {
  ToDoItem getMin() {
    int ans = heap[0];
    return new ToDoItem(new Date(…),
                        ans.description);
  }
}
```

What about `deleteMin`?

```
public class ToDoPQ {
   …
   ToDoItem deleteMin() {
     ToDoItem ans = heap[0];
     … // algorithm involving percolateDown
     return ans;
}
```

- Does not create a “red arrow” because object returned is no longer part of the data structure
- Returns an alias to object that was in the heap, but now it is not, so conceptual “ownership” “transfers” to the client

Less copying: use immutability

- (Deep) copying is one solution to our aliasing problems
- Another solution is **immutability**
  - Make it so nobody can ever change an object or any other objects it can refer to (deeply)
  - Allows "red arrows", but immutability makes them harmless
- In Java, a `final` field cannot be updated after an object is constructed, so helps ensure immutability
  - But `final` is a "shallow" idea and we need "deep" immutability

This works

```
public class Date {
   private final int year;
   private final String month;
   private final String day;
}
public class ToDoItem {
   private final Date date;
   private final String description;
}
public class ToDoPQ {
   void insert(ToDoItem i){/*no copy-in needed!*/}
   ToDoItem getMin(){/*no copy-out needed!*/}
   …
}
```

Notes:

- `String` objects are immutable in Java
- (Using `String` for `month` and `day` is not great style though)

This does *not* work

```
public class Date {
   private final int year;
   private String month; // not final
   private final String day;
   …
}
public class ToDoItem {
   private final Date date;
   private final String description;
}
public class ToDoPQ {
   void insert(ToDoItem i){/*no copy-in*/}
   ToDoItem getMin(){/*no copy-out*/}
   …
}
```

Client could mutate a `Date`'s `month` that is in our data structure

- So must do entire deep copy of `ToDoItem`

`final` is shallow

```
public class ToDoItem {
   private final Date date;
   private final String description;
}
```

- Here, `final` means no code can update the `date` or `description` fields after the object is constructed
- So they will always refer to the same `Date` and `String` objects
- But what if those objects have *their* contents change?
  - Cannot happen with `String` objects
  - For `Date` objects, depends how we define `Date`
- So `final` is a "shallow" notion, but we can use it "all the way down" to get deep immutability

This works

- When deep-copying, can “stop” when you get to immutable data

- Copying immutable data is wasted work. Such unnecessary copies is poor style

```
public class Date { // immutable
   private final int year;
   private final String month;
   private final String day;
   …
}
public class ToDoItem {
   private Date date;
   private String description;
}
public class ToDoPQ {
   ToDoItem getMin(){
    int ans = heap[0];
    return new ToDoItem(ans.date, // okay!
                        ans.description);
   }
}
```

What about this?

```
public class Date { // immutable
    …
}
public class ToDoItem { // immutable (unlike last slide)
    …
}
public class ToDoPQ {
  // a second constructor that uses
  // Floyd's algorithm
  void PriorityQueue(ToDoItem[] items) {
     // what copying should we do?
     …
   }
}
```

To copy or not to copy?

- Array
- `ToDoItem` object
- `Date` object

Homework 4

- You are implementing a graph abstraction
- As provided, `Vertex` and `Edge` are immutable
  - But `Collection<Vertex>` and `Collection<Edge>` are not
- You might choose to add fields to `Vertex` or `Edge` that make them not immutable
  - Leads to more copy-in-copy-out, but that's fine!
- *Or* you might leave them immutable and keep things like "best-path-cost-so-far" in another dictionary (e.g., a `HashMap`)

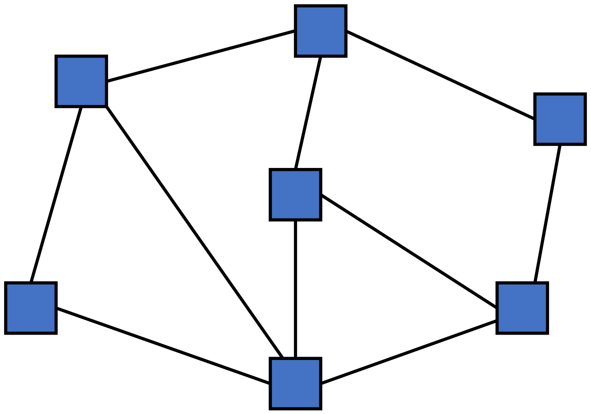
There is more than one good design, but preserve your abstraction

- *Great practice with a key concept in software design*

Spanning Trees

Spanning Trees

- Goal: Given a *connected* undirected graph **G**=(**V**,**E**), find a minimal subset of edges such that **G** is still connected
  - A graph **G2** = (**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



Observations

1. Any solution to this problem is a tree
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected
   - So **|E| >= |V|-1**

4. A tree with **|V|** nodes has edges
   - So every solution to the spanning tree problem has edges

Motivation

A **spanning tree** connects all the nodes with as few edges as possible

- Example: A "phone tree" so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

- Example: Electrical wiring for a house or clock wires on a chip
- Example: A road network if you cared about asphalt cost rather than travel time

This is the **minimum spanning tree** problem

  - Will do that next lecture, after intuition from the simpler case

Two Approaches

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

2. Iterate through edges; add to output any edge that does not create a cycle

Spanning tree via DFS

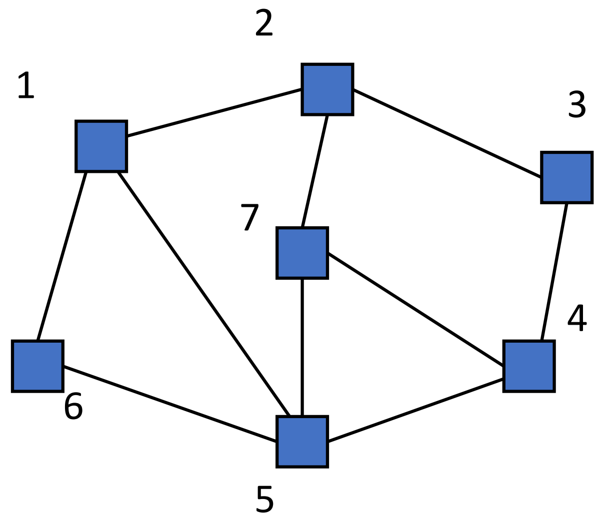
```
spanning_tree(Graph G) {
  for each node i: i.marked = false
  for some node i: f(i)
}
f(Node i) {
  i.marked = true
  for each j adjacent to i:
    if(!j.marked) {
      add(i,j) to output
      f(j) // DFS
    }
}
```

Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|E|)$

Example: Approach #1

Stack



Output:

Second Approach

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
    - Else it would have created a cycle
- The graph is connected, so we reach all vertices

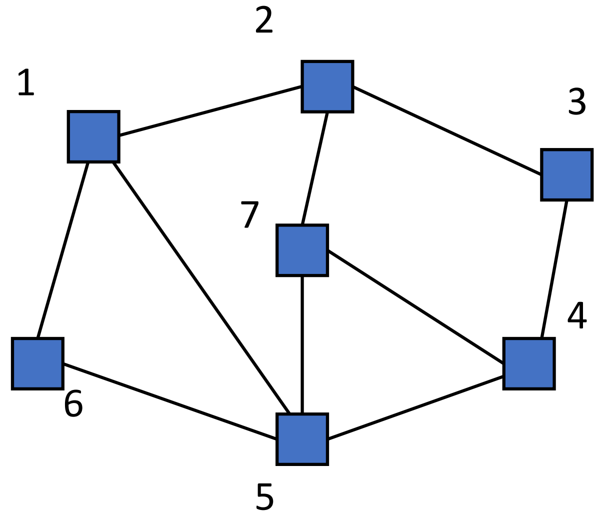
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

Example: Approach #2

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output:

Practice with Design Decisions

Graphs Edition!

Our three-eye-alien friend uncovered an impressively complete and up-to-date family tree tracing all the way back to the ancient emperor Qin Shi Huang. The alien wants to find a descendant of this emperor who's still alive, and could use your advice!

(According to Wikipedia, Qin Shi Huang had ~50 children, wow!)

What data structure would you recommend?
Why?

What algorithm would you recommend?
Why?