We’ll be covering two major topics in recitation today:

- **Priority Queues/Heaps** - which you have seen in lecture
- **Backtracking** - which will help you understand your programming assignment better

### Priority Queues & Heaps

- A *priority queue* is simply like a queue except that the next element to be dequeued always has the maximum priority.

- A priority queue is an *abstract data type* - which means that we just know that we can have any data structure beneath it, all we know is that we have a way to get the element of maximum priority from it.

- Since min-heaps are more common than max-heaps, we usually think of the element with the least integer value being the element with maximum priority.

- A heap is an *implementation* for a priority queue.

- We can see that we’ll need one important invariant for heaps - the element which will be accessed/ deleted from it must have minimum value. To maintain this invariant, while inserting or deleting from the priority queue, we need to do some special operations.

- As mentioned in lecture, for now, we’ll think of an array that stores the value in the heap, such that the children of the element at index $i$ are at $2i$ and $2i + 1$.

  *Notice that this means that we can’t use the index 0 of the array!*

- We’ll also see some important concept that are very common while dealing with operations on complex data structures.

  1. The invariant’s that hold for the operation as a whole (which are true at the beginning and at the end of the operation) are not necessarily true while the operation is being performed. So we are permitted to *temporarily violate invariants* while performing the operation and then restore them at the end.

  2. However, this also means that we need to find variations of the original invariant that hold true while the operation is being performed and reason that this will imply the stronger invariant that is true at the end of the operation.

Now let’s look at some of the code from lecture, especially to discuss the `is_heap_except_up` function that you didn’t write in lecture

(I’m sorry this code isn’t very neat today - I’m still trying to figure out how to do this in LaTeX!)
bool is_heap(struct heap_header* H) {
    if (!(H != NULL)) return false;
    //@assert \length(H->data) == H->limit;
    if (!(1 <= H->next && H->next <= H->limit)) return false;
    for (int i = 2; i < H->next; i++)
        //@loop_invariant 2 <= i;
        if (!(priority(H, i/2) <= priority(H, i))) return false;
    return true;
}

/* H is a valid heap except possibly at n, looking up in the tree */
bool is_heap_except_up(heap H, int n) {
    if (H == NULL) return false;
    //@assert \length(H->data) == H->limit;
    if (!(1 <= H->next && H->next <= H->limit)) return false;
    for (int i = 2; i < H->next; i++)
        //@loop_invariant 2 <= i;
        
        if (i != n && !(priority(H, i/2) <= priority(H, i))) return false;
        if (i/2 == n && (i/2)/2 >= 1 && !(priority(H, (i/2)/2) <= priority(H,i)))
            return false;
    return true;
}

void pq_insert(heap H, elem e)
//@requires is_heap(H) && !pq_full(H);
//@ensures is_heap(H);
{
    int n = H->next;
    H->data[n] = e;
    (H->next)++;
    int i = n;
    while (i > 1 && priority(H,i) < priority(H,i/2))
        //@loop_invariant 1 <= i && i < H->next;
        //@loop_invariant is_heap_except_up(H, i);
        { swap(H->data, i, i/2);
            i = i/2;
        }
    //@assert is_heap(H);
    return;
}
• Notice that is_heap_except_up(H, 1) is equivalent to is_heap(H). This fact is important when we’re showing that the loop invariant for pq_insert() implies the post-condition. We’ll do this at the end of recitation, if time permits.

• Now, let’s prove that is_heap_except_up(H, i) is indeed a loop invariant for pq_insert()
Backtracking

This is important for doing “Peg Solitaire Lab”, this week’s programming assignment. If you haven’t started yet, today would be a good time to start!

- Here is Wikipedia’s definition -
  Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c (“backtracks”) as soon as it determines that c cannot possibly be completed to a valid solution

- There are three common ways of searching for solutions while backtracking - breadth-first, depth-first and best-first

- We often visualize these as a tree, but note that we don’t actually construct a tree. Consider the tree that I’m about to draw on the board.

- **Breadth-first**: Here we traverse a particular level before going to the next level. In the example, this is the order:

- **Depth-first**: Here we simply go as deep as possible in the tree and if we don’t find our desired solution, we come back up and follow the other another branches in a similar manner. In the example, this is the order:

- **Best-first**: This follows a heuristic, which is a kind of rule that helps us estimate how likely it is that a particular branch will lead us to a solution.