- Vectors are basically resizeable arrays
- Vectors have a capacity. They reserve some extra memory where new elements are added

\[
\text{append(4)} \quad (3, 7) \rightarrow (3, 7, 4) \quad \text{Size} = 3 \quad \text{Capacity} = 5
\]

- Once capacity is exceeded the following is done:
  1. A new memory block with (2x) larger capacity is allocated
  2. The old data is copied over into the new block
  3. The old block is deallocated

\[
\text{append(7)} \quad (3, 7, 4, 6) \rightarrow (3, 7, 4, 6, 7)
\]
Suppose we create pointers that point to elements of a vector. What are the risks of doing that?

Suppose we fill a vector by appending \( m \) times. How many operations do we have to do asymptotically (in Big O)?

Suppose we create a vector of vectors of integers (vector<vector<int> >). Suppose that:

1. We add \( m \) integers in total to subvectors
2. All subvectors will end up non-empty
3. The subvectors can be added at any time

What is the worst case complexity (Big O of the number of operations) if in the top-level vector reallocation we:

1. Move only the metadata of the lower-level vectors
2. Move the whole contents of the lower-level vectors
Deques

- Deques are resizeable sequences like vectors, but internally they behave differently.
- Deques store elements in (reasonably large) equal-size chunks and when capacity is exceeded they simply add a new chunk.
- The data is more consecutive than in a list, but less than in a vector.
- As a result in practice it’s much faster than a list, but somewhat slower than a vector.
- The main benefit over a vector is that no reallocation is performed, so pointers to its data don’t get invalidated.
Deques

Exercises

- How would you keep track of where chunks are situated?
- How does the memory overhead of a deque compare to that of a vector?
A heap is simply a balanced binary tree where the value of a node is always greater or equal to the values of its children.

Example:

The root always contains the maximum element.
Let’s number the nodes from top to bottom and left to right:

Do you notice a pattern? Given node index $i$, what are the indices of its children?

How many levels (up to down) does a heap with $n$ nodes have?
To insert an element, we first create a new node according to the aforementioned numbering:
Next look at a path from that node to root:
Finally move the node up along this path, so the path would be decreasing:

Why is the result of those operations still a heap?
What is the worst-case time complexity of insertion?
Let’s look at some heap:

How would we delete the root from it?
First swap the root node with the last one:
Next we delete the last node:
Next we look at the "max-path" (path that always picks the highest value child):
Finally move the root down along the path until it has no children with greater value:

- Why is the result still a heap?
- What is the time complexity of deletion?
You can place a heap in an array, all structural information can be derived from indices:
Example

```c
int heap[MAX_HEAP_SIZE];
int heap_size;
void insert(int value) {
    int i = heap_size;
    heap_size++;
    heap[i] = value;
    while(i > 0) {
        int pi = (i−1)/2; // This is division rounded down
        if(heap[pi] < heap[i]) {
            swap(heap[pi], heap[i]);
        } else {
            break;
        }
        i = pi;
    }
}
```
Heap Exercises

How would you perform deletion in a location other than root?

What about changing the value of a random node?

How would you sort an array using a heap? What would be the time complexity?