Dynamic arrays, priority queues and distjointed sets

This week i have worked on a few additional data structures. While the last module focused on some elementary ones like linked lists, arrays, stacks, queues and trees this one involved

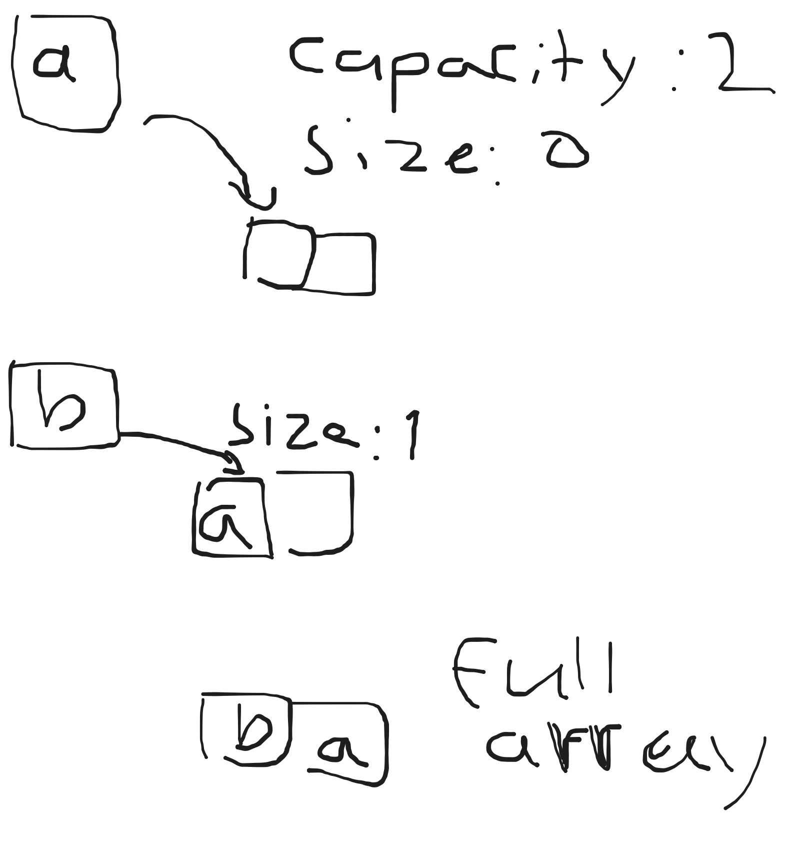
- Dynamic arrays
- Priority queues
- Disjointed sets

Dynamic arrays

The array from the previous module was static. The dynamic array is capable of decreasing and increasing in size as the array is operated on. The following operations are implemented in the dynamic array.

```
get(i): returns the element at location i

set(): set the element i to val*

pushBack(val): Adds val to the end of array

Remove(i): Removes element at location i

Size(): returns number of elements
```

Consider two pushBack operations on an emtpy array.



Once b is pushed back on the array, the array is full. Now that it is a dynamic array we can increase the size of this array with some constant factor for example 2.

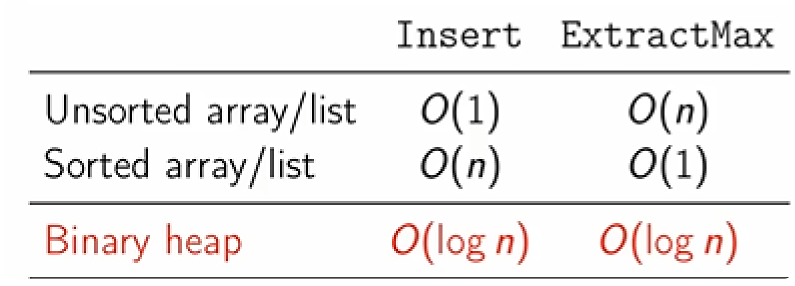
Amortized costs While it may be expensive to increase the capacity of the array, since we then have to move each element it is considered a rare operation. The idea of amortized costs is that if we take into account that expensive operations are rare, it creates a more nuanced view of the efficiency of the operations such as pushBack().

Priority queues

Priority queues is a data structures that have the following operations

- `getMax()` : O(1), simply return the root
- `insert(x)` : O(log n), insert at a leaf and “sift up”
- `extractMax()` : O(log n), remove root and “sift down”
- `changePriority()` : O(log n)

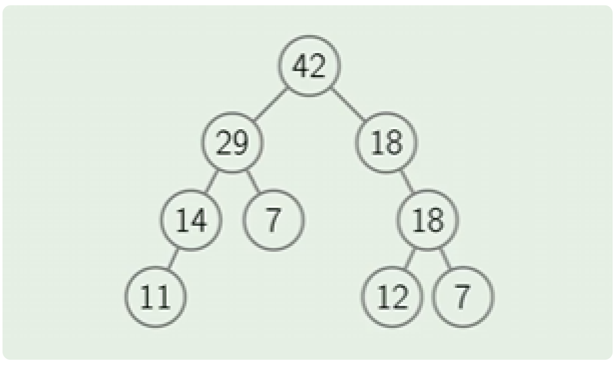
There is a broad range of algortihms that use priority queues including:

- Djikstra's algorithm: finding shortest path in a graph
- Prim's algorithm: constructing a minimum spanning tree of a graph
- Huffman's algorithm: constructing an optimum prefix-free enconding of a string
- Heap sort: sorting a given sequence 

	Insert	ExtractMax
Unsorted array/list	$O(1)$	$O(n)$
Sorted array/list	$O(n)$	$O(1)$
Binary heap	$O(\log n)$	$O(\log n)$

  Priorities queues can simply be implemented by both sorted and unsorted arrays, but that leaves one of the operatons inserting or extracting the max priority O(n). There exists another way to achieve greater runtime and that is to use the binary heap.

The binary heap to efficiently implement priority queues The binary heap is a tree where each parent have 0,1 or 2 child nodes and each parent is atleast the value of the children.



The **operations** on binary max heap

- getMax(): taking the root O(1)

- insert(): we insert the item at a leaf and swap it with the parent if it's larger than it's parent. The running time is O(tree height).
- extractMax(): O(tree height)
- changePriority(): O(tree height)
- siftUp() and siftDown() are helper functions to help us restore balance in the tree after extacts, and change priority
- remove():

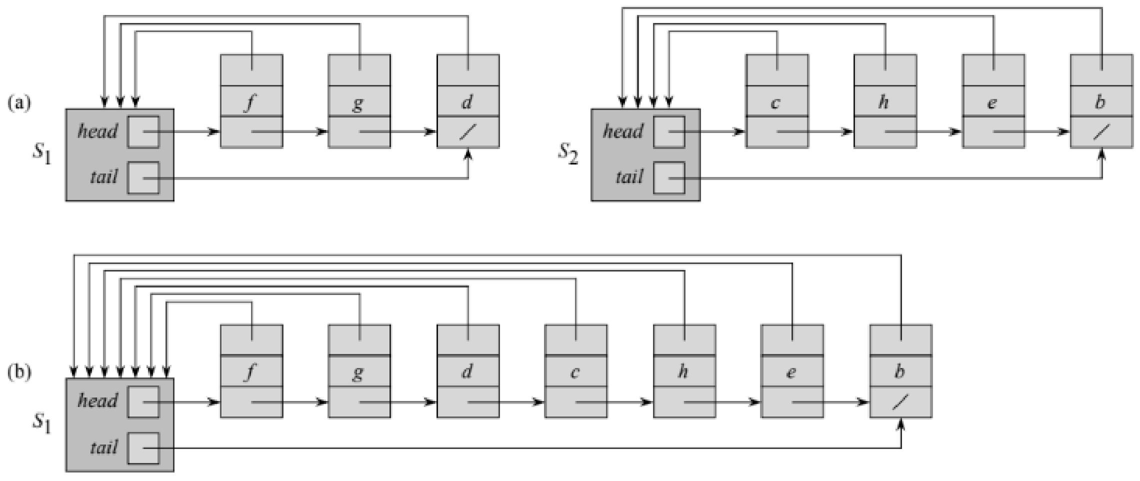
The binary heap can also be used to for implementing the heap sort algorithm.

Disjointed sets

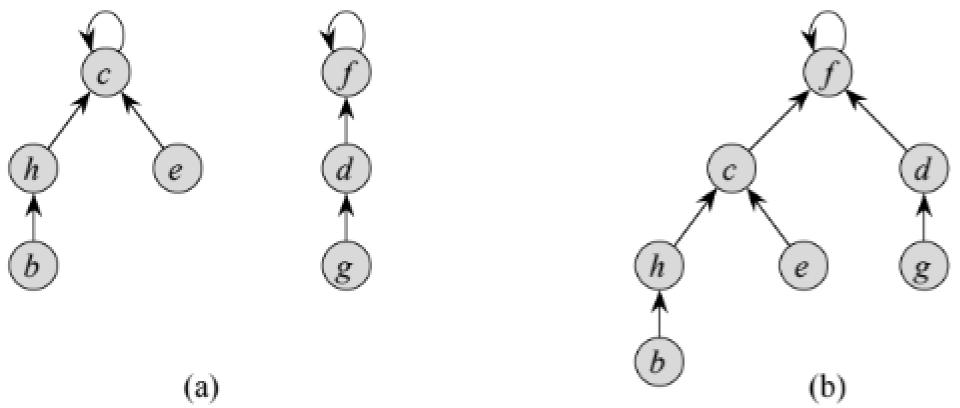
Finally, this module covered a **disjoint-set data structure**. This data structure maintains a collection S = {S1, S_2....,S_k} of disjoint dynamic sets. The operations on the disjointed set are

- makeSet(x) which creates a new set whose only memeber (and thus **representative**) is *x*. Since the sets are disjoint, we require that *x* not already be in some other set.
- union(x,y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of the two sets. We assume that the two sets are disjoint prior to the operation.
- find-set(x) returns a pointer to the representative of the (unique) set containing x.

Implementation using linked list One way to implement this data structure is to use a linked-list. Here is shown a union operation where the S_2 is appended to S_1. We can use the *tail* pointer to find quickly where to append the second set, namely after *d*.



Implementation using disjoint-set forest The disjoint-set forest shown here below shows how such implementation could look like. The two sets are first shown in figure *a* as trees each node pointing to it's parent node, and the root pointing to itself. Figure (b) shows the union(g,e). To do that the representative for the set (c,h,e,b) that is c, the root if we mentally viewed as a tree is union with f the root of the second set by fixing it's head pointer to the representative of the f set. The tail pointer in the set (f,g,d) is then moved all the way down to b.



(a) (b)

The running times of several operations on this implementation can be improved using two **heuristics**. The first one is **union by rank**. For each node we maintain a rank, which is an upper bound on the height of the node. In union by rank, we make the root with smaller rank point to the root with larger rank. If the ranks are equal we arbitrarly choose one of the roots as parent and increment its rank.

The second heuristic is **path compression**. It's used during findSet operations to make each node on the find path point directly to the root.

The intuition here is that if we union the smaller tree to the larger one, then we will keep the tree as *shallow* as possible. Path compression is applied whenever we want to find a set x, we store the information about the nodes visited on the path to the root. This can make the operations "almost constant time" .

Working with these data structures has deepened my understanding of algorithm design.

- Dynamic arrays showed me how theoretical inefficiencies (resizing) can be balanced through amortized analysis.
- Priority queues demonstrated the importance of choosing the right implementation for performance-critical applications.
- Disjoint sets highlighted how small optimizations (union by rank, path compression) can transform performance.

Each structure gave me both new technical skills and a broader appreciation for how algorithms are carefully engineered to balance simplicity, flexibility, and efficiency.

References

- *Data Structures and Algorithms Specialization – Module 2*
- Cormen et al., *Introduction to Algorithms*, Chapters 6 (Heapsort) and 21 (Disjoint Sets)

Disjoint-set Data Structures CS 225 | Disjoint Sets