## Containers:

| | Access | | | Insert | | | Append | | | Delete | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | θ | O | Ω | θ | O | Ω | θ | O | Ω | θ | O | O |
| Array | 1 | 1 | 1 | 1 | n | n | Amortized O(1) | | | 1 | n | n | n |
| | Array supports random access in constant time. | | | Best case insert at end, worst at beginning. Need to copy/reallocate. | | | Occasionally needs to copy/reallocate with O(n) | | | Same reasoning as inserting. | | | Doubles in size when reallocating, may be wasteful. |
| LL | 1 | n | n | 1 | n | n | n | n | n | 1 | n | n | n |
| | Best case access at root, else need to traverse sequentially. | | | Given a pointer to where to insert; need to traverse to find prev. | | | Always need to traverse sequentially and find the end. | | | Given a pointer to the deleted node; need to traverse to find prev.[1] [2] | | | Requires memory for pointers, otherwise matches n. |
| LL (Tail) | 1 | n | n | 1 | n | n | 1 | 1 | 1 | 1 | n | n | n |
| LL (Doub) | 1 | n | n | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n |

| | Push | | | Pop | | | Top/Front | | | Size | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | θ | O | Ω | θ | O | Ω | θ | O | Ω | θ | O | O |
| Stack (Arr) | Amortized O(1) | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n |
| Use base and top ptr. | We may need to grow, and copy/reallocate. | | | Popping the top done by decrementing top ptr. | | | Dereference base ptr. | | | Pointer arithmetic between base and top ptr. | | | |
| Stack (LL) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n | n | n |
| Use head ptr. | Change head ptr to point at new node. | | | Delete head ptr, point to next. | | | Dereference head ptr. | | | Constant all around if size is stored. Best case if empty list, null ptr. | | | |
| Queue (Arr) | Amortized O(1) | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n |
| Circ. arr, front/back ptr | Set back ptr's element, then point to next. Reallocate when front = back. | | | Set front ptr to point to next element. | | | Dereference front ptr. | | | Pointer arithmetic between front and back. | | | |
| Queue (LL) | 1 | n | n | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n | n | n |
| Use head ptr, front. | Constant all around if with tail pointer. | | | Delete head ptr, point to next. | | | Dereference head ptr. | | | Constant all around if size is stored. Best case if empty list, null ptr. | | | |

| | Insert | | | Search Max | | | Delete Max | | | Creation | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | θ | O | Ω | θ | O | Ω | θ | O | Ω | θ | O | O |
| PQ (Arr) | Amortized O(1) | | | 1 | n | n | 1 | n | n | 1 | 1 | 1 | n |
| Unsorted array. | Simple push_back into a vector. | | | Linear search. | | | Linear search. | | | Any container would work. | | | |
| PQ (Arr) | 1 | n | n | 1 | 1 | 1 | 1 | 1 | 1 | 1 | nlogn | nlogn | n |
| Sorted array. | Binary search O(logn), but inserting takes O(n). | | | Access the most extreme element, already sorted. | | | Access the most extreme element, already sorted. | | | Need to sort container prior to usage. | | | |
| PQ (Heap)[3] | 1 | logn | logn | 1 | 1 | 1 | 1 | logn | logn | 1 | n | n | n |
| Use array based heap. | Insert at bottom of tree, fixUp. | | | Look at the front of array or top of heap. | | | Can only remove root. Move last into root, fixDown. | | | Heapify. | | | |
| PQ (ArrLL) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | N/A | N/A | N/A | n |
| Array of linked lists. | Only works when there is a restricted amount of priority levels. | | | | | | | | | | | | |

| | Insert | | | Search | | | Delete | | | Creation | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | θ | O | Ω | θ | O | Ω | θ | O | Ω | θ | O | O |
| HT (Perf.)[5] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | n | n | N/A | N/A | N/A | n |
| | Hash to integer, go to address, find nothing, insert. | | | Go to address, find it, return. | | | Go to address, mark it unoccupied. | | | | | | |
| HT (SC)[6 7] | 1 | $\alpha$ | n | 1 | $\alpha$ | n | 1 | $\alpha$ | n | N/A | N/A | N/A | N |
| M linked list for each addr. | Go to address, search if it exists in linked list. Insert if it doesn't. | | | Go to address, search linked list. Worst if all hashed to same list. | | | Go to address, search linked list. Worst if all hashed to same list. | | | | | | |
| HT (LP)[8] | 1 | $\sim\alpha$ | n | 1 | $\sim\alpha$ | n | 1 | $\sim\alpha$ | n | N/A | N/A | N/A | N |
| Array with linear probing. | Avg. time for miss will depend on load factor. Worst case, ~1 load factor. | | | Worst case, ~1 load factor with a cluster until the very end of table. | | | Worst case, ~1 load factor with a cluster until the very end of table. | | | | | | |
| HT (DH)[9] | Amortized O(1) | | | 1 | $\sim\alpha$ | n | 1 | $\sim\alpha$ | n/2 | N/A | N/A | N/A | N |
| SC/LP with dynamic hashing. | Once load factor threshold reached, double size of table, rehash everything. | | | Worst case, a cluster as big as threshold allows. | | | Worst case, a cluster as big as threshold allows. | | | | | | |

**Sorts**[4]

| Algorithm | Complexity | | | Notes | Memory | Notes |
|---|---|---|---|---|---|---|
| | Ω | Θ | O | | O | |
| Bubble | n | n^2 | n^2 | Compares adjacent items, move the maximum all the way to right. Repeat. | 1 | n^2/2 comparisons, n^2/2 swaps |
| Bubble (A) | n | n^2 | n^2 | Adds a check if no swaps were made, stop. | 1 | If input is sorted, just do 1 pass. |
| Selection | n^2 | n^2 | n^2 | Find smallest, swap with first, find second smallest ... | 1 | n^2/2 comparisons, n-1 swaps in best/avg/worst case. |
| Selection (A) | n | n^2 | n^2 | Don't swap if item is in correct position. | 1 | 0 swaps best case. Minimal copying, good for small arrays. Cheap in swaps, expensive comparisons. |
| Insertion | n | n^2 | n^2 | Consider elements one at a time, insert to proper place among considered. | 1 | n^2 comparisons, n^2/2 swaps worst case, n^2/4 average. Good for partially sorted data. |
| Counting | n | n | n | First pass counts records that go into each bucket. Second pass computes address, third copies. | n | Fast but needs more memory. |
| Heapsort | nlogn | nlogn | nlogn | Heapify, then remove elements one at a time, filling original array from back to front. | 1 | Unstable. |
| Quicksort | nlogn | nlogn | n^2 | Select pivot, then partition. Repeat. | 1 + stack logn | Unstable. |
| Mergesort | nlogn | nlogn | nlogn | Partition, and then merge step by step. | n | Stable. |

## Trees[10]

| | Insert | | | Search | | | Delete | | | Parent | Child | Memory | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | ϴ | O | Ω | ϴ | O | Ω | ϴ | O | O | O | Ω | O |
| BT (Arr) | 1 | n | n | 1 | n | n | 1 | n | n | 1 | 1 | n | 2^n |
| | Worst case if the tree is full. | | | Worst case if deleting last element, of it doesn't exist. | | | Worst case if deleting last element, of it doesn't exist. | | | Simple formulas to calculate index of parent and children. | | Best case if tree is evenly distributed, worst if "stick". | |
| BT (Ptr) | 1 | N | n | 1 | n | n | 1 | n | n | n | 1 | n | n |
| Pointer based nodes. | Worst case if the tree is full | | | Worst case if deleting last element, of it doesn't exist. | | | Worst case if deleting last element, of it doesn't exist. | | | To find parent, must traverse from root. Else store parent ptr in node. | | Only uses space as needed, no waste except for ptr mem. | |

| | Insert | | | Search | | | Delete[12] | | | Sort | | | Memory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ω | ϴ | O | Ω | ϴ | O | Ω | ϴ | O | Ω | ϴ | O | O |
| BST[11] (Ptr) | 1 | logn | n | 1 | logn | n | 1 | logn | n | nlogn | nlogn | n^2 | n |
| | Traverse tree for the proper place to insert. | | | Worst case a "stick". Can be represented with O(h). Avg. is balanced tree. | | | Find the node to delete. Find the inorder successor. O(h). | | | Insert n items, which is O(nlogn). Then inorder traversal, O(n). | | | |
| AVL[13] | 1 | logn | logn | 1 | logn | logn | 1 | logn | logn | nlogn | nlogn | nlogn | n |
| Adelson-Velskii Landis | Insert as normal, compute balance factors, rebalance if necessary. | | | Worst case "stick" is not possible, logn worst case. | | | Delete like a BST, then rebalance. | | | Insert n items, which is O(nlogn). Then inorder traversal, O(n). | | | |

## Graphs

- A simple (no self loops) directed graph with V vertices may have V(V-1) edges.
- A graph is sparse when $E \ll V^2$ *or* $E \approx V$, dense when $E \approx V^2$.
- We can have adjacency list (implemented using array of linked lists), adjacency matrix, or distance matrix. Matrices are n^2 memory.
- Depth-first search uses stacks, Breadth-first search uses queues. Both can find a path from one to another. Queues will give shortest path if all costs are equal.

## Path Finding

| | DFS (Adjlist) | DFS (Adjmat) | BFS (Adjlist) | BFS (Adjmat) |
|---|---|---|---|---|
| θ | V(1 + E/V) = (V + E) | V^2 | V(1 + E/V) = (V + E) | V^2 |
| O | V + V^2 | V^2 | V + V^2 | V^2 |
| | Visits every vertex at most once, O(V). Then visits adjlist, at most once, and each contains average E/V edges, O(1+E/V). | Visits every vertex at most once O(V), then visits adjmat row at most once, O(V). | Visits every vertex at most once, O(V). Then visits adjlist, at most once, and each contains average E/V edges, O(1+E/V). | Visits every vertex at most once O(V), then visits adjmat row at most once, O(V). |
| | Use adjlist when graph is sparse, hence $E \approx V$ and complexity is O(2V). If it is dense, $E \approx V^2$, use adjmat because adjlist DFS will be O(V + V^2), worse than O(V^2). | | | |

## Minimum Spanning Trees

- Given a weighted, undirected graph. Find subgraph such that all vertices are connected and the sum of all edges are minimal. Cannot be a cycle, hence it must be a tree.
- Shortest edge and second shortest edge must be included in every MST. Third one depends, if it produces a cycle, then no, else yes.
- Greedy algorithms to solve this problem: Prim's[14] and Kruskal's[15]. Dijkstra's for shortest path is very similar to Prim's.

| | Prim's (Linear) | Prim's (Heap/PQ) | Kruskal's |
|---|---|---|---|
| θ | V(V + 1 + V) = (V^2) | VlogV + ElogV | ElogV + E + V |
| O | V^2 | VlogV + V^2logV | V^2logV + E + V |
| | Outer loop through every V. Inner loop, finds smallest O(V), set visited O(1), loop through neighbors which is O(V). | Outer loop is O(V). Inner loop, pop PQ O(logV), set visited O(1). This is VlogV. Step 3 runs E times, and adding to PQ is logV. | Sort edges, O(ElogE). Worst case, O(ElogV^2) = O(ElogV). Loop over E edges, add/test/discard, O(logE) or O(logV). |
| | Use the linear version for a dense graph. If it's a dense graph, $E \approx V^2$, and O(VlogV + V^2logV) is worse than O(V^2). If sparse, use heap/PQ version. | | For a dense graph, we can do O(ElogV) of Prim's or Kruskal's. Use Kruskal's it's always faster. |

## Traveling Salesman Problem

- Backtracking solves constraint satisfaction, and branch and bound solves optimization.
- Hamiltonian cycle: cycle that traverses each node exactly once, and no vertex except first/last may appear twice.
- TSP: Hamiltonian cycle with least weight.

$if \lim_{n\to\infty} \frac{f(n)}{g(n)}\ is < \infty$, then f(n) is O(g(n)).

1. LL deletion can be improved by: passing a pointer to the previous node, or by overwriting data in target node with next node data, and then delete next node.
2. Best Destructor: careful when just deleting one node, set next as null.
   `~Node(){delete next;} ~Linked List() {delete headPtr;}`
3. Heaps: use binary trees as underlying structure, often implemented using arrays. A tree is heap ordered (max) if the key at each node is not less than the keys of all the node's children.
   a. Children nodes found by 2i and 2i + 1, parent node found by floor(i/2), [ i is 1 indexed ].
4. Internal Sort: "I can see all elements" – file fits into memory, random access available.
   Indirect Sort: "I can see all indices" – reorder indices instead of items, important when copying is expensive.
   External Sort: "Too many elements" – items to be sorted are on disk, accessed sequentially or in blocks.
   Stability: preservation of relative order of items with duplicate keys in a file.
5. Perfect hashing can be done if the set is never going to change, e.g. 50 words, C++ language. No collisions.
6. Load factor $\alpha = \frac{N}{M}$ where N keys are placed in an M-sized table. We select $M \approx N$ so that load factor is 1, for O(1) complexities..
7. In separate chaining, α is the average number of items per linked list, can be > 1.
8. In linear probing, α is the percentage of table positions occupied, and must be <= 1. Runtimes can be improved by using quadratic probing, but it may never find some addresses due to residual squares. Both are types of open addressing.
9. Dynamic hashing: double the size of table when it fills up, e.g. α = 0.5. Expensive but infrequent. This ensures complexities of hash table does not become too big. This technique can be used with any of the above collision resolution schemes.
10. Binary tree: ordered tree in which every node has at most two children. Can be proper, in which all nodes have exactly 0 or 2 children, and/or complete, in which every level is full, with possible exception of the last level, filled left to right.
11. Binary search tree: all left children are less than parent, all right children are greater than parent. Ties can go either direction.
12. If deleted node has left OR right child, simply replace deleted node with child.
    If there is both left AND right child, find inorder successor (node that comes right after in an inorder traversal), or smallest child of the deleted's right child. Transplant inorder successor to the deleted node.
13. AVL trees self balance to mitigate O(n) worst case operations. Each node records its height, and computes its balance factor:
    `height(n->left) – height(n->right)`
    If balance exceeds absolute value of 1, it must be balanced. Done through rotations, a local change involving only three pointers and two nodes.
    We need a double rotation if signs disagree.
    If node balance > 1, we rotate right. If left node balance is negative, we rotate left there first. Vice versa.

14. Prim’s algorithm:
    a. Repeat until every node is visited:
        i. From the set of vertices which is unvisited, select the one with smallest minimum distance, v.
        ii. Set v as visited.
        iii. For each unvisited vertex w adjacent to v, test whether its current minimum distance is greater than distance of v to w. If so, set minimum distance of w, and set its previous as v.
15. Kruskal’s algorithm:
    a. Sort all edges. O(ElogV)
    b. Loop through all edges: O(E)
        i. Check if it produces a cycle:
            1. Union-find: every disjoint set should have its unique representative, O(VαV), or O(V).
            2. If they are in the same set, compare representatives.
        ii. Add. O(1)