Algorithms and Data Structures

Algorithms

Complexity Analysis

Measuring Complexity

In complexity analysis, the number of the most expensive (time consuming) operation is used as a proxy for the complexity of the algorithm as a whole. The underlying assumption is that each such operation takes the same amount of time (e.g. addition of two numbers takes constant time). This will be true if both numbers can fit into one computer word: 32 bits, number < 2^{32}. Fortunately, most algorithms do not deal with such large numbers, so counting operations usually suffices.

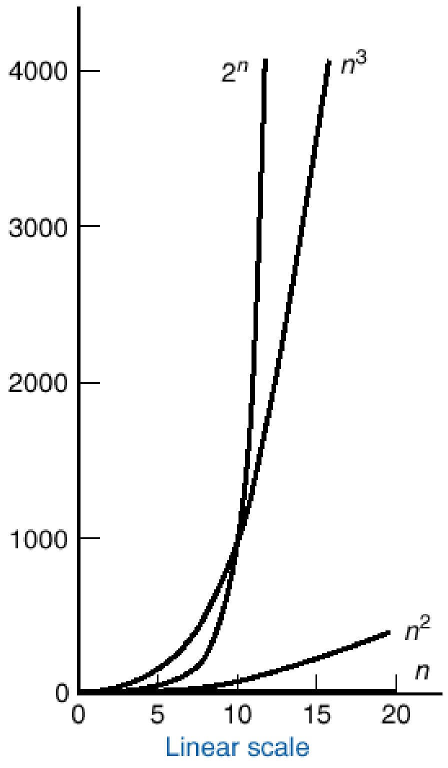
Big-O Notation

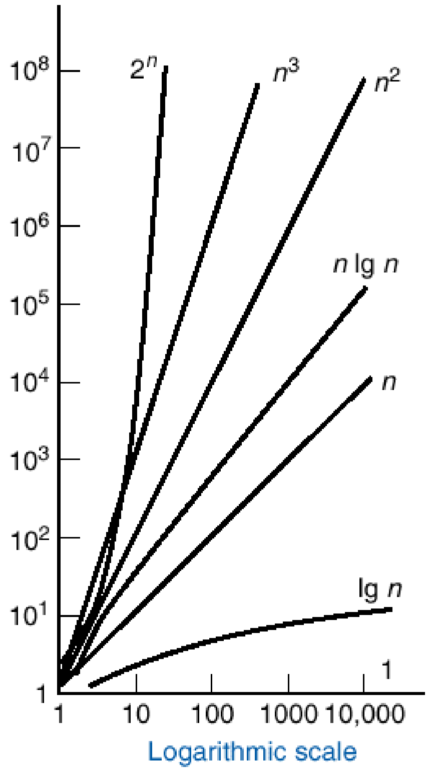
For two functions f(n) and g(n), we say that f(n) is in O(g(n)) if there are constants c and N, such that f(n) < cg(n) for all n > N. In other words, f(n) is O(g(n)) if f(n) grows no faster than g(n) multiplied by some constant. Every function is a member of a large number of O(x) classes, however we are usually only interested in the smallest one: for example, if a function is O(x^2) then it is also O(x^3)

If a program is in stages: stage 1 operates on m inputs, is linear O(m), and stage 2 operates on n inputs, is linear O(n). Then the whole program is O(m) + O(n) = O(m+n). If m << n, then O(n). Alternatively, if the program operates on each of n inputs m times, program is O(m*n)

Big-Omega and Big-Theta

In addition to Big-O notation, we also have Big-Omega and big-Theta notation. Big-Omega is simply the opposite of Big-O; it represents a lower bound rather than an upper bound. Big-Theta is upper and lower bound. For example, if f(n) is O(x), then it always grows no faster than x times a constant. If f(n) is Ω(x), then it always grows no slower then x times a constant. If it is both, then we say f(n) is Θ(x), and it grows at the same rate as x (give or take a constant).





Complexity Classes

- O(1): Execute instructions once (or a few times), independent of input. Example: pick a lottery winner
- O(log n): Keep splitting the input, and only operate on one section of the input
- O(n): Execute instruction(s) once for each data item. Example: linear search
- O(n log n): split the input repeatedly, and do something to all the segments. Example: many sorting algorithms
- O(n^2): For each item, do something to all the others. Example: nested loops
- O(2^n): Exponential
- O(n!): Number of ways of arranging n items. Example: brute forcing passwords

Recurrence

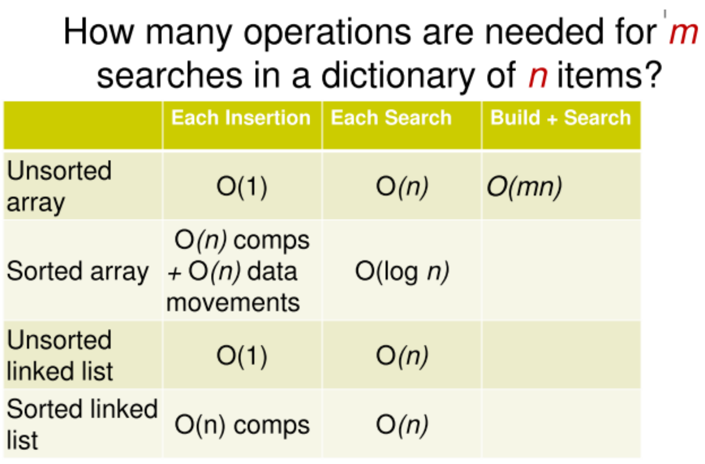
A recurrence relation is an equation that recursively defines a sequence. Recurrences appear a lot in many algorithms and data structures, especially divide-and-conquer algorithms.

Master Theorem

- The Master Theorem is a shortcut for determining the order of complexity of a recursive algorithm
- The theorem states that for an algorithm given by $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, with $f(n) = O(n^d)$:
    - $d > \log_b a\,,\ T(n) = O(n^d)$
    - $d = \log_b a\,,\ T(n) = O(n^d \log n)$
    - $d < \log_b a\,,\ T(n) = O(n^{\log_b a})$
- a is the branch factor, the number of splits of the problem made at each iteration
- b is the rate at which the size of each subproblem decreases
- So, for example in binary search, both of these would be two
- f(n) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems

Arrays and Pointers

- Arrays require random access memory - can access any array element at any time
- To initialise a string use 'char arr[] = "code"' or 'char *string = "code"'
- To pass an array to a function, use 'func(int **A, int n, int m)' or 'func(int A[][], int n, int m)'
- Pointers: int var is a variable, int *ptr = &var is a pointer to var. The value of *ptr is the address of var. You cannot do 'int *ptr = 7', as this is trying to store '7' in some unknown memory
- If you want to declare a pointer and then give it a value, best to use malloc, or define var i and then write 'int *ip3 = &i'



How many operations are needed for *m* searches in a dictionary of *n* items?

	Each Insertion	Each Search	Build + Search
Unsorted array	O(1)	O(*n*)	*O(mn)*
Sorted array	O(*n*) comps + O(*n*) data movements	O(log *n*)	
Unsorted linked list	O(1)	O(*n*)	
Sorted linked list	O(n) comps	O(*n*)	

Malloc

Malloc is a dynamic memory allocation function in C which returns a void pointer of specified size. A void pointer is not a pointer to void or to nothing; it's a pointer to memory, but not any data of any particular type. Whatever argument goes into malloc (e.g. int, char, etc), the pointer should be typed to the same thing with a star after it. For example:

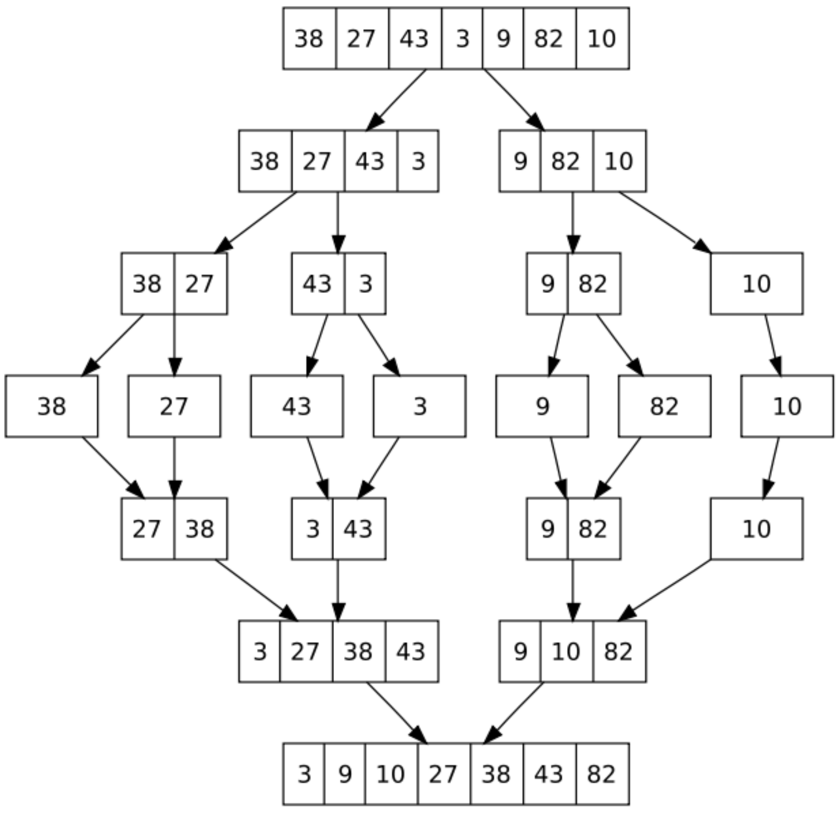
```
int *ptr;
ptr = malloc(10 * sizeof (*ptr));              /* without a cast */
ptr = (int *)malloc(10 * sizeof (*ptr));       /* with a cast */
```

z = (int*) malloc(sizeof(int)*10)
w = (int*) realloc(z, sizeof(int)*15)

Sorting Algorithms

Merge Sort

Merge sort works by taking two already-sorted lists and combining them into a single sorted master list. Where do the original sorted sublists can from? We can call merge sort recursively. All we need to do for this to work is pass an array containing the unsorted items, and two counter variables pointing to the first and the last elements of the array (see code).



Merge Sort Analysis

Advantages

- Easy to implement
- Guaranteed order nlog n
- Stable

Disadvantages

- Requires extra space
- Slower typical sort speed than quicksort

Merge Sort Code

```
merge(item C[], item A[], item [B], int n, int m)
{
        for(i=0,j=0,k=0;k<n+m;k++)  // k is a loop iteration variable, n and m are array sizes
        {
                if(i==n) // if at the end of list A, copy everything from list B
                {
                        C[k]=B[j];
                        j++;
                        continue;
                }
                if(j==m) // if at the end of list B, copy everything from list A
                {
                        C[k]=A[i];
                        i++;
                        continue;
                }
                if(A[i}<=B[j])  // if A[i] is smaller copy A
                {
                        C[k] = A[i];
                        i++;
                }
                else // if B[j] is smaller copy B
                {
                        C[k] = B[j];
                        j++;
                }
        }
}

mergesort(A,first,last)
{
        item C[last-first+1];  // this array just stores sorted data locally
        mid = (last-first+1)/2;  // picks a midpoint in the array A
        A = mergesort(A,first,mid);  // runs mergesort on the first and second halves separately
        B = mergesort(A,mid+1,last);
        C = merge(A,B);
        for(i=first;i<last;i++)
        A[i] = C[i];
}
```

Quicksort

Quicksort is a popular divide-and-conquer sorting algorithm with an average case n log n. The algorithm proceeds in the following steps:

- Pick an element, called a pivot, from the list
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation
- Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values
- The base case of the recursion are lists of size zero or one, which never need to be sorted

Implementation of Quicksort

- Pivot chosen as the last element in the list, although this results in poor performance with already-sorted lists
- The pivot can be moved around, but the value of the pivot always stays the same (i.e. it follows the pivot as it moves)
- The i pointer should begin at one space to the left of the first element of the list, while the j pointer should begin on the rightmost element
- Pre-increment both pointers on every pass of the central loop
- When comparing values to the pivot, a weak inequality ($\geq$) must be used, as otherwise values equal to the pivot will be left out of place
- After exiting the loop, swap the final resting place of the pointers with the right-most node (the pivot), so that it too is in order

Quicksort Analysis

Advantages

- Very fast, average nlog n
- In-place sort, no extra memory
- Stable

Disadvantages

- Order n^2 worst case
- Partition is difficult to code
- Requires random access to elements
- Not stable

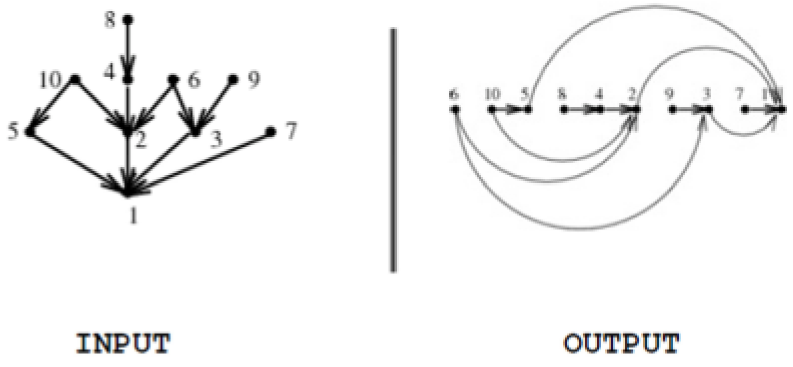
Quicksort Code

```
int partition(item A[], int l, int r) // left and right bounds
{
        int i = l - 1;
        int j = r;
        int v = A[r]  // pick the right-most element. Simple but not ideal
        while(1)  // repeat until break out of loop
        {
                while (A[++i]<=v)
                {
                        continue;  // if left value is less than pivot, do nothing
                }
        while (A[--j]>=v)
        {
            continue;  // if right value is more than pivot, do nothing
        }
        if(i>=j)
        {
            i=r  //reset i if it overshoots the end
        }
        break;  // exit loop when j and i overlap
        swap(A[i],A[j]);
    }
    swap(A[i],A[r]);   // put pivot in the correct position
    return(i);
}

void quicksort(item A[], int l, int r)
{
    if (r <= l)
    {
        return;  // do nothing if array has one element or less
    }
    i = partition(A,i,r);  // partition A into two sorted arrays
    quicksort(A,l,i-1);  // partition and sort each of those sub-arrays
    quicksort(A,i+1,r);
}
```

Topological Sort

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering. Such a topological sort is often not unique, because every node is not ordered with respect to every other pair.

Source removal is an algorithm that produces a partial sort by repeatedly removing one of the source nodes and placing it in the next free space in the list. There must be at least one source (smallest value) and one sink (biggest value) in order for the algorithm to work.



The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, that is $O(E + V)$, or roughly $O(V^2)$ for a dense graph.

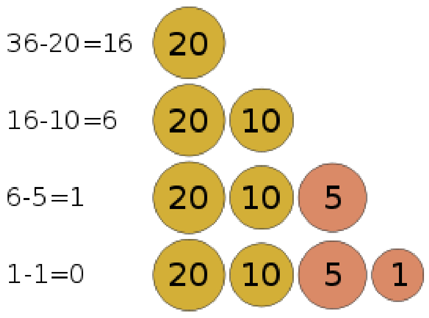
Shortest Path Algorithms

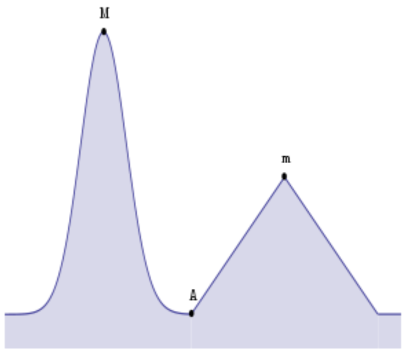
Shortest Path

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized. The brute force approach to solving this problem, to simply compare each route sequentially and pick the shortest, is $O(V!)$, which is completely intractable for even moderately sized graphs. We need a better approach.

Greedy Algorithms

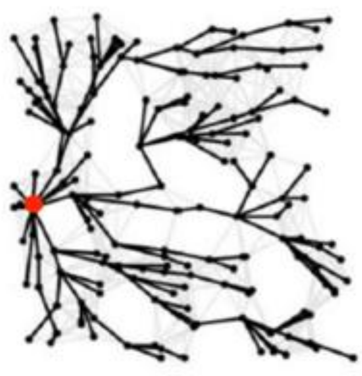
A greedy algorithm is a type of algorithm that takes a locally-optimal solution at each stage of the problem, in the hope that this will lead to a globally optimal result. The basic idea is to make whatever choice seems best at the moment and then solve the subproblems that arise later. Greedy algorithms can solve some types of problems (e.g. change-finding), but not others (e.g. finding the global maximum for functions with turning points).





Dijkstra's Algorithm

Dijkstra's Algorithm is a greedy algorithm for solving the shortest path problem. It is based on the key insight that if the shortest path from A to Y passes through X, then the subpath along this shortest path from A to X must also be the shortest path between A and X. In other words, any subpath along a shortest path is also a shortest path. The algorithm will fail if there are negative edge weights. The algorithm actually yields a shortest path tree, with is a spanning tree showing the shortest path from the source node to every other node in the graph.



Dijkstra's algorithm requires maintaining a record of the current shortest paths from the source node to every other node, say in a matrix called dist$[v]$, and also information concerning which node immediately proceeds every other node along the shortest path, say in prev$[v]$. In processing all the nodes, it is useful to maintain an ordered list of all remaining nodes to be visited, with order determined by estimated distance from the source node. This information can be stored in a heap, with the top node in the heap always indicating the next node to be visited (as it is the nearest).

Most of the cost of Dijkstra's algorithm relates to the cost of implementing the priority queue. If it is implemented using a heap, the total cost is $O\left((V+E)\log V\right)$, which is roughly $O(V^2 \log V)$ for a density connected graph.

Dijkstra's Algorithm Code

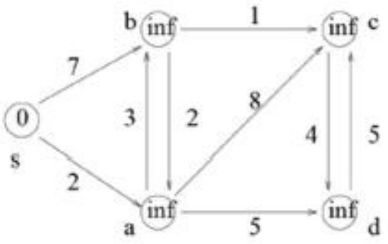
The first stage of the algorithm is simply to initialise all the distances to infinity.

```
initialize(G,V,s,pred,dist)
{
        int i;
        for(i=0;i<V;i++)
                dist[i] = MAX_INT;
        dist[s] = 0;
        for(i=0;i<V;i++) pred[i]) = NULL;
}
```

Next is the code for the main loop of the algorithm. While the priority queue is not empty, pick the top node from the queue (the closest node to the current node), and then for each node that can be reached from the new current node, if the path from the source straight to v is less than the path from the source to u and then from u to v, then update the dist and pred arrays, and also the priority queue, such that this new minimum path replaces the old path. Note that the priority queue must be kept sorted to reflect any changes in the path lengths.

```
run(G,V,s,pred,dist)
{
   pq_node_t  *pq;
   pq = makePQ(G); /* vertices into min PQ,
                dist  from s as in Graph, as key */
   while(!emptyPQ(pq))
   {
      u = deletemin(pq);
      for(/*each v reached from u */)
         if(dist[u] + edgeweight(u,v) < d[v])
               update(v,pred,dist,pq);
     }
}

update(v,pred,dist,pq)
{
        dist[v] = dist[u] + edgeweight(u,v);
        pred[v] = u;
        decreaseweight(pq,v,dist[v]);
}
```

Dijkstra's Worked Example

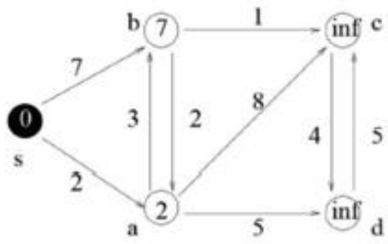


Step 0: Initialization.

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

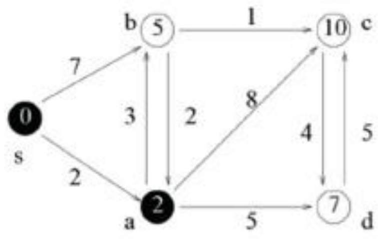


Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

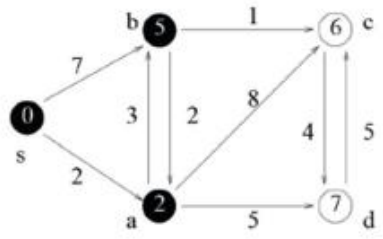


Step 2: After Step 1, a has the minimum key in th priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

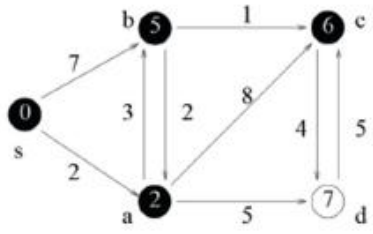


Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

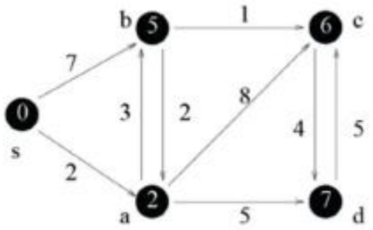


Step 4: After Step 3, c has the minimum key in the ority queue. As $Adj[c] = \{d\}$, work on d and up information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7



Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.

Floyd-Warshall algorithm
This is an extended version of the shortest path problem, in which one needs to find the shortest path from every node to every other node. One way to do this would simply be to run Dijkstra's algorithm once for every node, which would have $O(V^3 \log V)$ for dense graphs. An alternative is to use the Floyd-Warshall algorithm, which has $O(V^3)$. Another advantage of FW is that it will not give non-sensical answers in the presence of negative edges, as will Dijkstra's.

The Floyd-Warshall algorithm is best implemented using two matricies, one (say A) to contain the path lengths between all nodes, and one (say prev) containing the next node to pass through along the shortest path between any two nodes. The algorithm works by implementing a triple nested-loop over all of the nodes in A. For each possible intermediate node i, if the current minimum distance between s and t is greater than the distance going from s to t via i, then adopt this as the new minimum distance. Note that the prev matrix must also be updated accordingly (not shown in code).

```
for(i=0;i<V;i++)
  for(s=0;s<V;s++)
    for(t=0;t=V;t++)
        A[s][t]=(min(A[s][i] +A[i][t], A[s][t]);
```

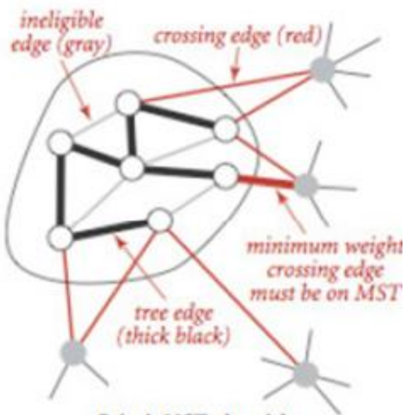
Minimum Spanning Tree

Minimum Spanning Tree
A minimum spanning tree is a subgraph that is: a tree (no cycles), contains every vertex (spans), and possesses the minimum sum of weights of all such subgraphs. Minimum spanning trees are distinct from shortest paths as the former must past through all nodes in the graph, whereas shortest paths do not. An MST must have exactly V-1 edges, as otherwise there will be a cycle. Greedy algorithms are good candidates to use in buildings MSTs: start with an MST, and then keep adding shortest connection edges that do not produce cycles. Minimal spanning trees for a particular graph are not necessarily unique.

Prim's MST algorithm
Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. It is the preferred method for dense graphs, and is easiest with matrix representation. The algorithm outputs a set of edges that corresponds to the minimal spanning tree.

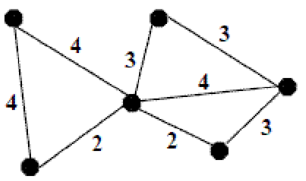
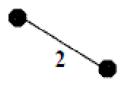
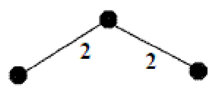
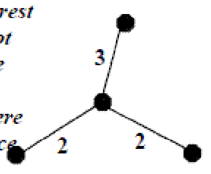
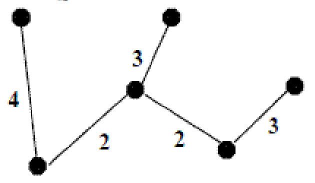
Steps of the algorithm:

- Pick any node at random (as all nodes will be reached by MST)
- Examine all nodes that are accessible from this node, and visit the node with the shortest path
- Examine all nodes crossing the current cut that have not already been visited, and visit the node with the light edge (the smallest path length)
- In the event of ties, pick at random (doesn't matter)
- Continue until every node has been visited

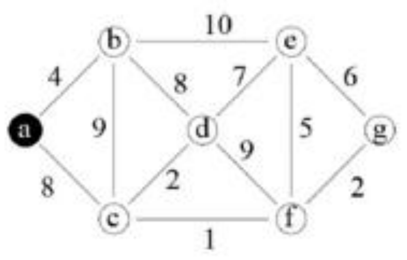


Prim's MST algorithm

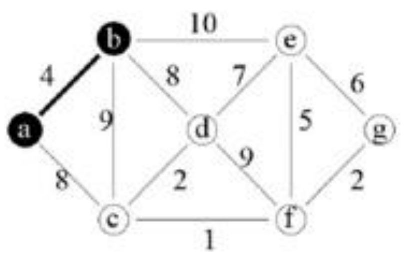
Prim's Algorithm

1 *Given a network............* 	2 *Choose a vertex*	3 *Choose the shortest edge from this vertex.* 
4 *Choose the nearest vertex not yet in the solution.* 	5 *Choose the next nearest vertex not yet in the solution, when there is a choice choose either.* 	6 *Repeat until you have a minimal spanning tree.* 

The fringe vertices are stored in a priority queue, so that the shortest can always be easily accessed. The existing MST is stored in a prev array which can be backtraced to find the original path. It is also necessary to store an array or list of nodes that have already been visited, so that they are not visited twice. If implemented using an adjacency matrix, it has $O(V^2)$.



Step 1.1 before
S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}



Step 1.1 after
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

Prim's Algorithm Code

```
prim(G,wt,root)
{
   for every u in V { key[u] = ∞; inmst[u] = FALSE;}
   dist[root] = 0; pred[root] = NULL;
   PQ = makePQ(V); /* all vertices in PQ */
   while(!empty PQ)
   { u = deletemin(PQ);
     for every (v adjacent to u)
     { if (inmst[v]=FALSE&& wt[u][v])< dist[v])
         {dist[v] = wt(u,v); /* update distance */
          decreasewt(PQ,v,dist[v]);/* update PQ dist*/
          pred[v]=u; /* update path information */}
     }
     inmst[u] = TRUE;
   }
 } /* at end: MST = {{v,pred[v]}: v in V - {root}} */
```

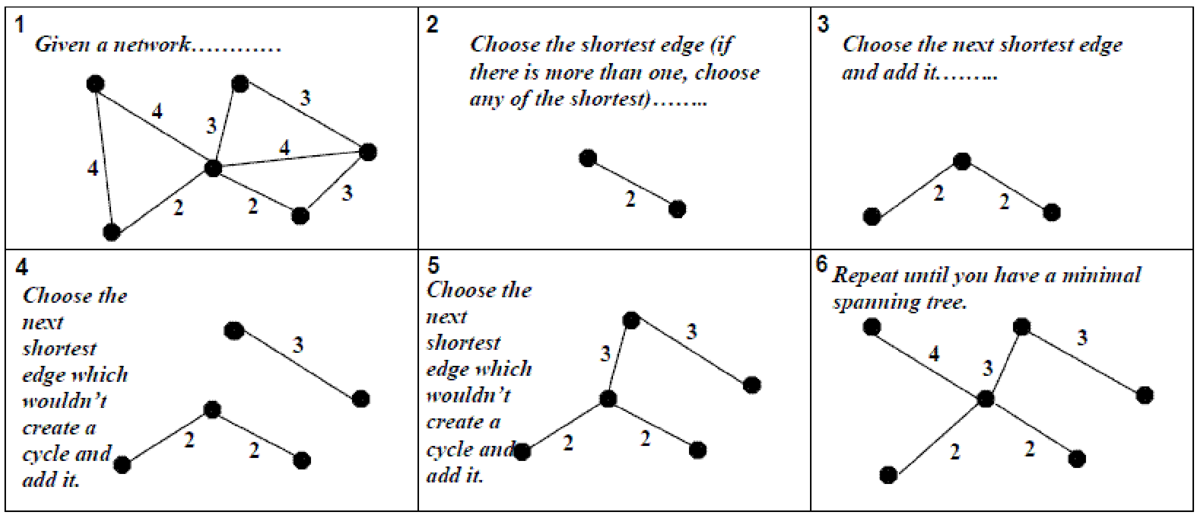
Kruskal's MST algorithm

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. It differs from Prim's MST algorithm in that it repeatedly adds edges, rather than repeatedly adding nodes to the MST. The next line is always the shortest (minimum weight) only if it does not create a cycle. Kruskal's algorithm runs in $O(E \log V)$, making it faster than Prim's.

Implementing this algorithm requires the use of a new data structure and algorithm: the disjoint-set data structure and the union-find algorithm.

Disjoint-set data structures have two key operations: find and union. Find: Which subset is an element in? Union: Join two subsets into a single subset.

Kruskal's Algorithm



1 *Given a network…………*

2 *Choose the shortest edge (if there is more than one, choose any of the shortest)……..*

3 *Choose the next shortest edge and add it……..*

4 *Choose the next shortest edge which wouldn't create a cycle and add it.*

5 *Choose the next shortest edge which wouldn't create a cycle and add it.*

6 *Repeat until you have a minimal spanning tree.*

Data Structures

Graphs

Features of Graphs

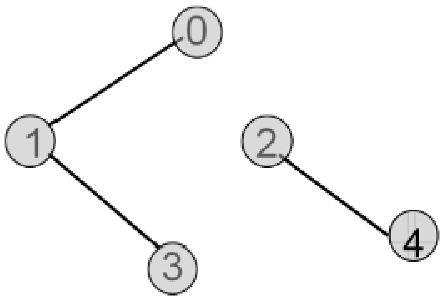
A graph is a set of vertices and edges, with edges between selected vertices. Edges can have direction and/or weight, but this is not essential. Dense graphs have most nodes connected to most other nodes, in which case $E \approx V^2$, and hence $O(V+E) \to O(V^2)$.

Types of Graphs

- Undirected graphs have no direction associated with the edges - they are bidiretional
- Directional graphs do have a direcdtion associated with the links
- Connected graphs have each pair of vertices connected to each other, either directly or via one or more intermediate vertices
- A complete graph is one where every vertex is connected directly to ever other vertex. It will always have $V(V-1)/2$ edges
- A tree is simply a special type of graph. Specifically, it is a graph that is connected (every node reachable from every other node), and acyclic (you can never get back to your starting node without backtracking)

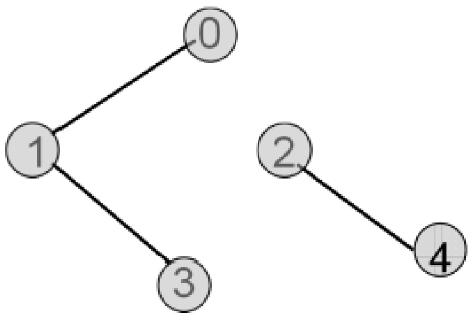
Representing Graphs

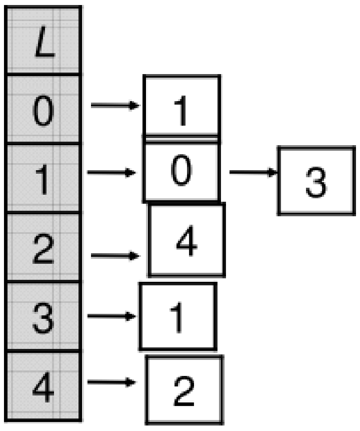
One way of representing graphs is by an adjacency matrix, which is a $V \times V$ matrix with ones indicating a connection between two nodes (the 'first' node is always down the rows, the 'second' node across the colums). For undirected graphs, the adjacency matrix is always symmetric, as any forward link must also be a backward link. Adjacency matrices are easy and quick to access, but are wasteful of space if the graph is sparsely connected (and hence most elements are zero). Weighted graphs can also be represented in this way by simply placing the weights in the matrix elements.



A	0	1	2	3	4
0		25			
1	25			40	
2					10
3		40			
4			10		

An alternative representation is an adjacency list, which consists of an array (or list) of lists, one list for each node in the graph. The linked list for every node consists of all the nodes that are connected to that original node, in the order in which they are connected. Adjacency lists are of size order $V+E$, compared to V^2 for adjacency matrices. This means that in general adjacency lists are better for sparsely connected graphs.



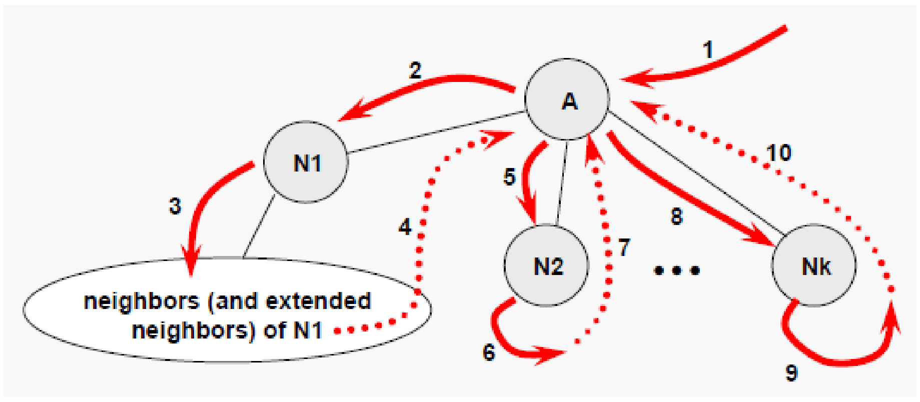


Traversing Graphs

Graph traversal is the process of systematically visiting every node in a graph, often along with reporting the value of the node. Tree traversal is a special case of graph traversal, and is generally simpler because there can be no issues of cycles, and all nodes in a tree must be connected. There are two main classes of graph traversal algorithms: depth-first and breadth-first.

Depth-First Search

To ensure that each node is visited only once, the traversal algorithm needs to mark each node as 'visited'. This may most simply be done by creating a 'visited[n]' array, which simply stores a binary value for each node indicating whether it has been visited or not. Depth first search begins at some initial node A, then visits all the neighbours of A beginning with N1, recursively visiting all the neighbours of N1 (and so on down the chain), until there are no more neighbours. Then it jumps back to A, and proceeds to visit the remaining unvisited neighbours of A (in this case moving on to N2).



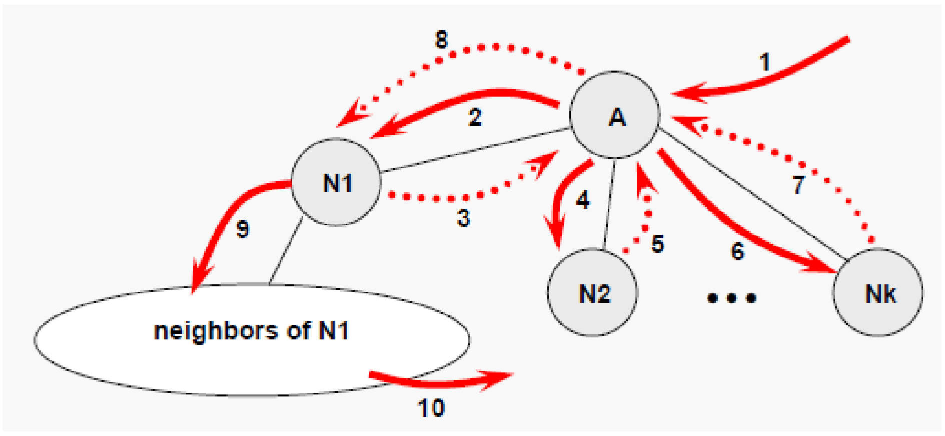
There are three distinct orderings in which depth-first traversal can be performed:

- Preorder traversal: (i) Visit the root, (ii) Traverse the left subtree, and (iii) Traverse the right subtree. Preorder traversal of the above tree: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10
- Inorder traversal: (i) Traverse the left most subtree starting at the left external node, (ii) Visit the root, and (iii) Traverse the right subtree starting at the left external node. Inorder traversal of the above tree: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- Postorder traversal: (i) Traverse all the left external nodes starting with the left most subtree which is then followed by bubble-up all the internal nodes, (ii) Traverse the right subtree starting at the left external node which is then followed by bubble-up all the internal nodes, and (iii) Visit the root. Postorder traversal of the above tree: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

```
void preorder(node_t   *t)
{
          if(t==NULL) return();
                    visit(t);   /* visit first */
                    preorder(t->left);
                    preorder(t->right);
}
void inorder(node_t   *t)
{
          if(t==NULL) return();
                    inorder(t->left);
                    visit(t);   /* e.g. print value */
                    inorder(t->right);
}
```

Breadth-First Search

A breadth first search works differently do a depth-first search, moving 'across' rather than 'down' the hierarchy of nodes. Beginning at node A, a breadth-first search will first visit N1, then N2, and so forth through Nk, until all immediate neighbours of A are visited. Then it will proceed to traverse all the unvisited immediate neighbours of N1, then traverse the immediate neighbours of N2, … Nk in similar fashion. In this way it moves across the hierarchy visiting one 'layer' at a time, until the entire graph is visited.



Priority Queue

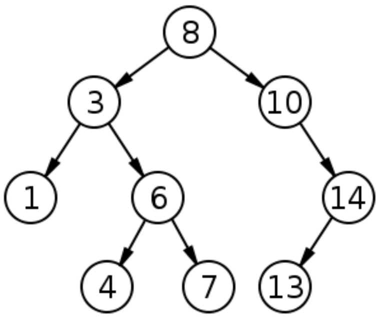
A priority queue is an data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. An element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue. Priority queues can be implemented in either lists or arrays.

One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority element is retrieved first. Stacks and queues may be modelled as particular kinds of priority queues. In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; thus, the first element inserted is always the first retrieved. To improve performance, priority queues typically use a heap as their backbone, giving O(log n) performance for inserts and removals, and O(n) to build initially.

Trees and Lists

Binary Trees

A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right". Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right. A binary search tree is a special type of binary tree in which the left child (and all its children) always has a key less than its parent node, and the right child (and all its children) always has nodes greater than (or equal to) the parent.

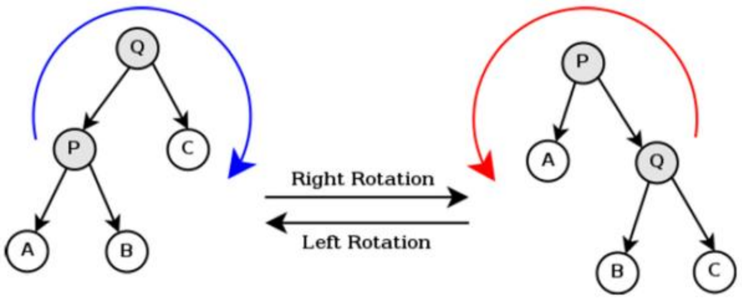


For a complete binary tree of n nodes, to get to the bottom will take not more than $\log_2 n$ key comparisons, compared to n for a linked list. Average case for insertion/search/deletion in a binary search tree is also log n. The worst case for a binary search tree is a stick, when all items are inserted in sorted or reverse sorted order, the list degenerates to a linked list - with O(n) search/insert times.

Balancing Binary Trees

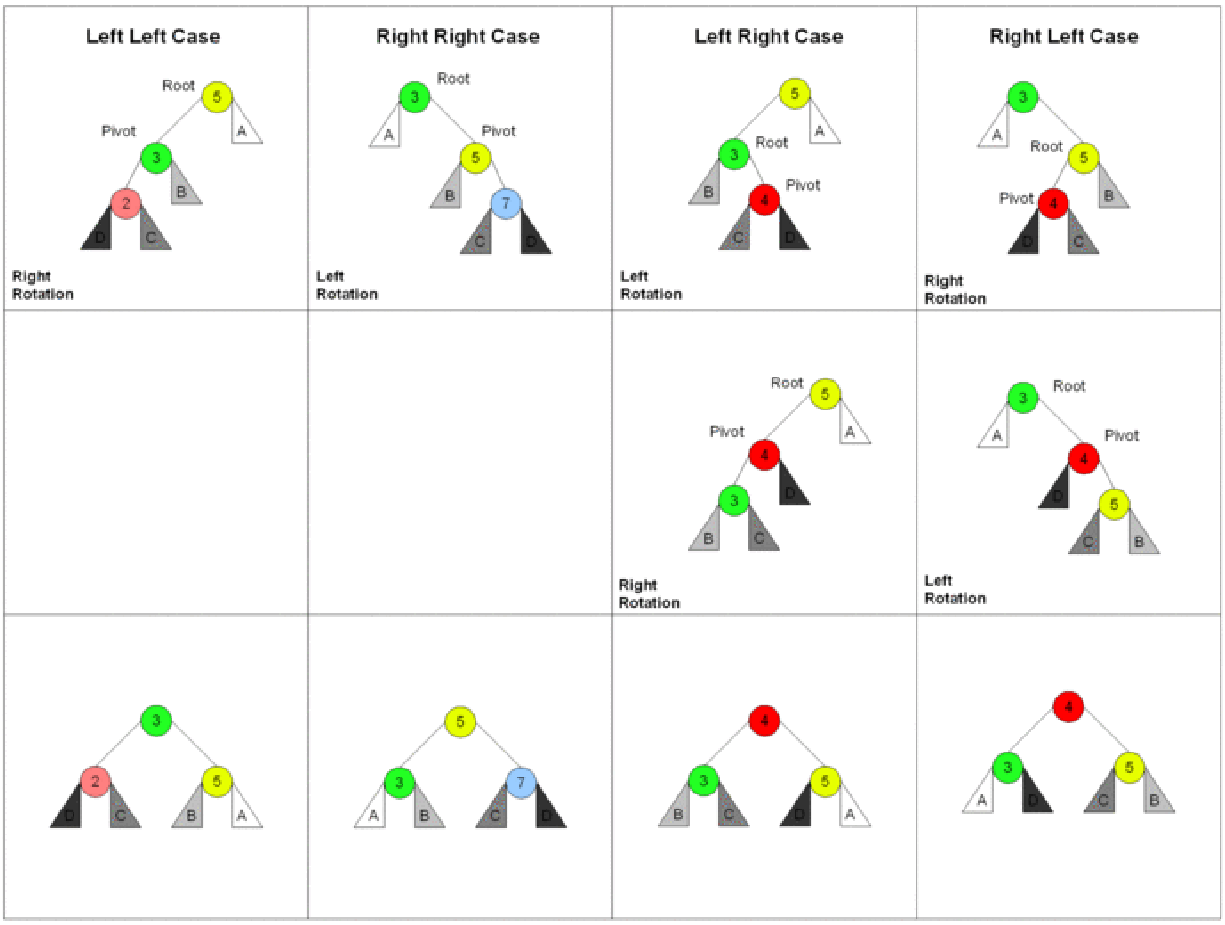
The major drawback to binary search trees is that the code to balance them after addition or deletion of new elements is very difficult. A tree is said to be balanced if no left and right subtrees of any node differ in height by more than one. The method of balancing is known as tree rotation. A tree rotation moves one node up in the tree and one node down, which allows the tree to be 'more balanced' (hence improving performance), without interfering with the order of the elements in the tree.

When balancing, the direction of rotation is always taken starting at the new root node (the pivot), and curving around the old root node.



Code for Tree-Balancing

```
RotateR(node P, node Q, grandparent)
        par = Q;
        child = P;
        Q->left = P->right;
        P->right = Q;
        grandparent->r/l = P;
```

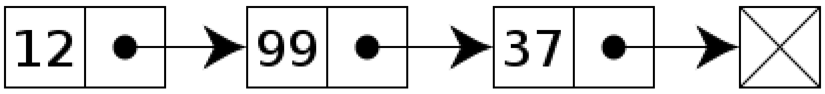


Other Types of Trees

- Splay tree: a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again
- 2-3-4 tree: a tree where every node with children (internal node) has either two, three, or four child nodes
- AVL tree: a special self-balancing type of tree

Linked Lists

a linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a reference (in other words, a link) to the next node in the sequence.



The main advantage of linked lists is that insertion if much quicker than for arrays, as we can just change to pointers, and don't have to shift the whole array. A disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values, because the storage overhead for the links may exceed by a factor of two or more the size of the data. Also, linked lists do not have random access of nodes.

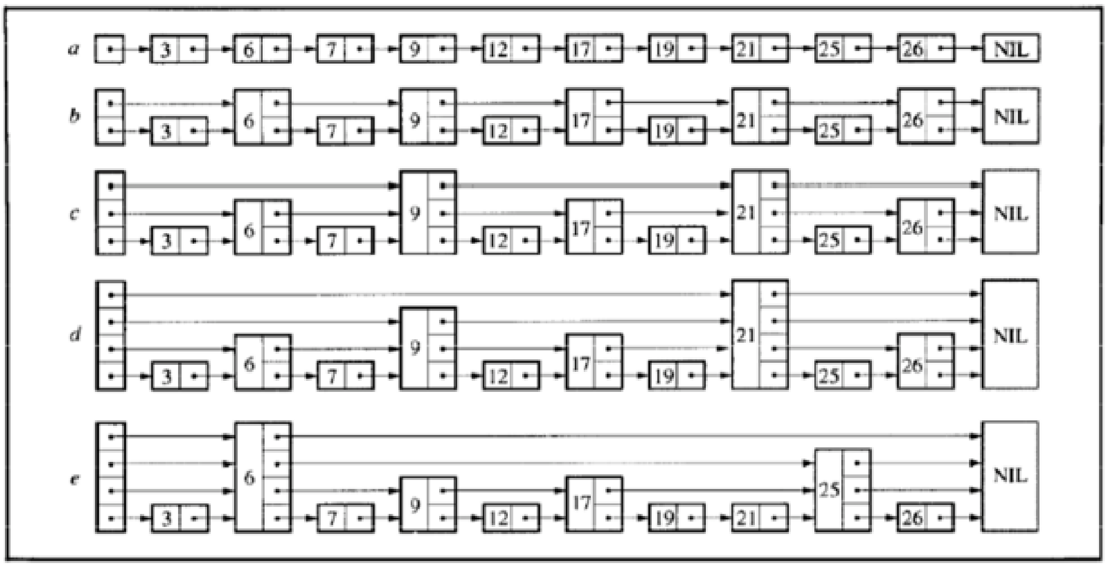
Code for Nodes

```
typedef struct node_tt node_t; // create alias; 'node_t' means 'struct node_tt'
        // note: it is necessary to define node_tt first because node_t is a recursive structure, so when
        we actually define the structure we already need something to refer to, otherwise the
        compiler won't know what we are taking about

struct node_tt  // define node_t structure
{
        int key;
        char value[MAX_STRING];
        node_t* next;
} node_tt;

typedef struct // define list_t structure (note: not recursive)
{
        node_t* head;
        node_t* foot;
        int size;
} list_t;
```

Skip Lists

A skip list is a data structure for storing a sorted list of items using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees. The head node always has MAXLEVEL levels, and each subsequent node as some probability p of adding an additional level beyond the base level.

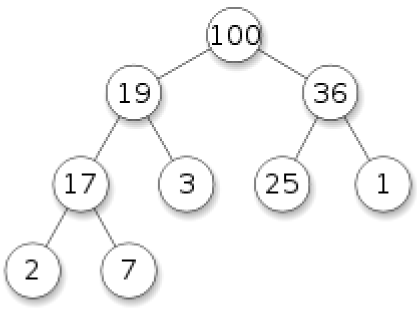
They are very useful because they have the easy insert/delete capability of a linked list, and also the log n binary search capability of a sorted array. Also, unlike binary trees, they are not especially difficult to code. The main disadvantage is that they are a probabilistic data structure, meaning that they have O(n) behaviour as a worst case with extremely small probability.



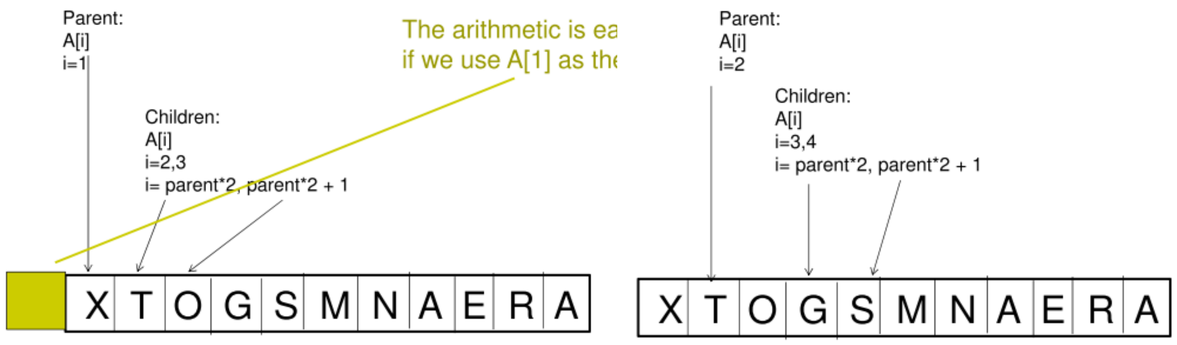
Heaps

The Basics

A heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap (usually either greater than or less than). Unlike a binary tree, there is no ordering between sister or cousin nodes; only between parent and child nodes. The maximum number of children each node can have depends on the type of heap, but in many types it is at most two, which is known as a "binary heap". Heaps must also be complete trees, which means that every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Heaps can be instantiated in arrays, as indicated in the diagrams shown below.



Creating a Heap

There are two potential methods for creating a heap. The simplest method is to simply insert the items one-by-one into an array, and then upheap() as each new item is inserted. Since there are n items and the complexity of upheap() is $\log n$, the overall complexity of this method is $n \log n$. The second method is to first insert all items randomly into an array (unordered), and then apply downheap() level by level, beginning at the base of the tree and moving upwards. This process for building a heap is $O(n)$.

Upheap

Used to add an element to a heap, whilst retaining heap properties. Complexity is $O(\log n)$. Consists of three steps:

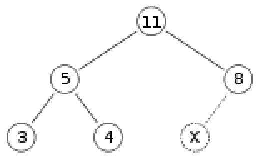
- Add the element to the bottom level of the heap
- Compare the added element with its parent; if they are in the correct order, stop
- If not, swap the element with its parent and return to the previous step

Upheap Code

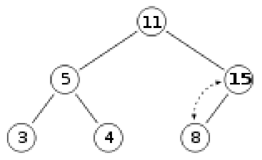
```
upheap(int k)
        v = A[k];
        A[0] = INT_MAX; /* sentinel */
        while(A[k/2] <= v) /* note integer arithm */
                A[k] = A[k/2];
                k = k/2;
        A[k] = v;
```

Upheap Example

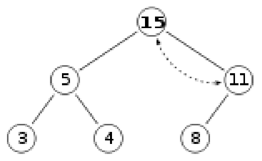
As an example, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 is greater than 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



However the heap property is still violated since 15 is greater than 11, so we need to swap again:



Downheap

Used to delete the root node (highest priority) from the heap, whilst retaining the heap properties. Complexity is $O(\log n)$. It consists of three steps:

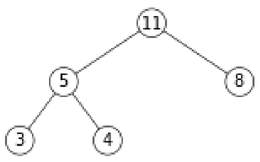
- Replace the root of the heap with the last element on the last level
- Compare the new root with its children; if they are in the correct order, stop
- If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap)

Downheap Code

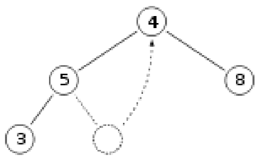
```
downheap(int k)
        v = A[k];     /* value, or priority */
        while(k<=n/2)  /*A[k] has children*/
        { /* compare with largest child */
                j= k+k;
                if(j<n && a[j]<a[j+1])
                        j++;
                if (v>= a[j])
                        break;/* heap OK */
                a[k] = a[j]; k = j;}
        a[k] = v;
```

Downheap Example

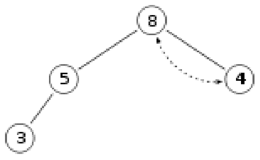
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



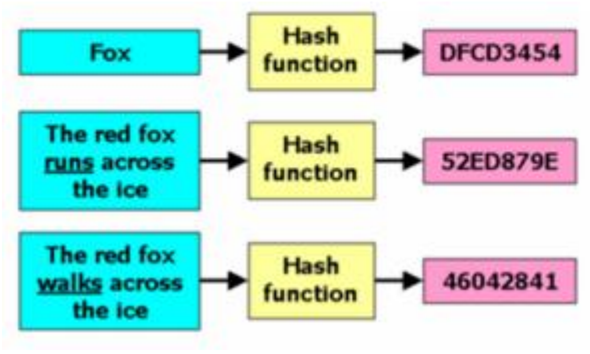
Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:

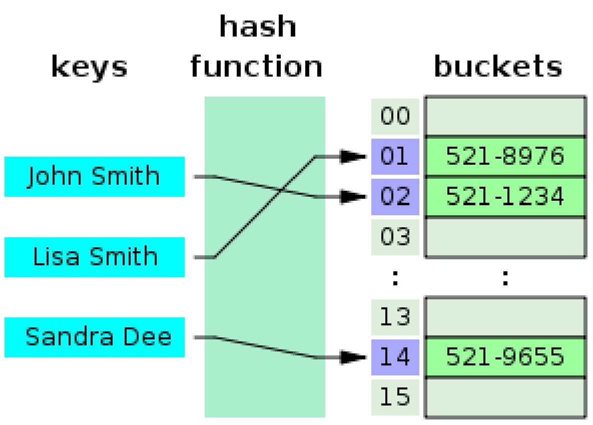


Hash Tables

The Basics

A hash table is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index based on the key value, allowing information to be stored in an array. Ideally, the hash function will assign each key to a unique bucket, but this ideal situation is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

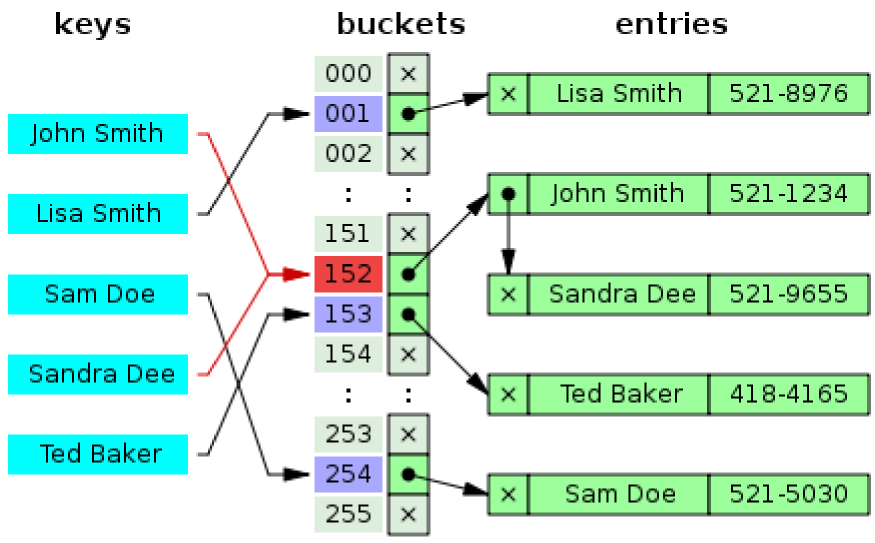




Where array SIZE < number of records, there will definitely be collisions. Where array SIZE > number of records, there will often still be collisions, and we must handle them. Good hash functions have fewer collisions, but we can't ever assume there will be none.

Open Hashing (Separate Chaining)

Each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation. The list structure need not be particularly efficient, because each list should only hold at most a small number of items - otherwise the hash table as a whole is working poorly.



```
insert(HT,item)
{
        new node = /* make a list node */
        /* put item in the list node */
         index = hash(item->key);
        if(HT[index]==NULL) HT[index]=newnode;
        else
        {       newnode->next = HT[index]->node;
                HT[index] = newnode;
        }
```

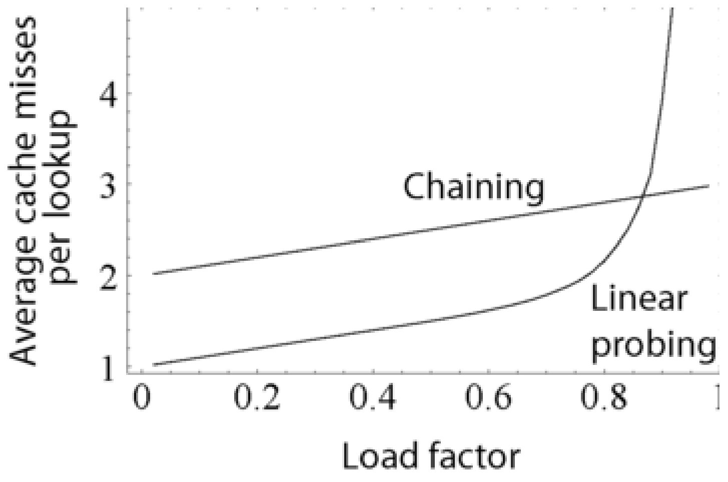
Open hashing is somewhat risky because if many items happen to hash to a single location, or if the number of items is much bigger than the table, then performance will degrade, though it does so gracefully (see graph below).

Linear Probing

If there is a collision, put the item in the next available slot

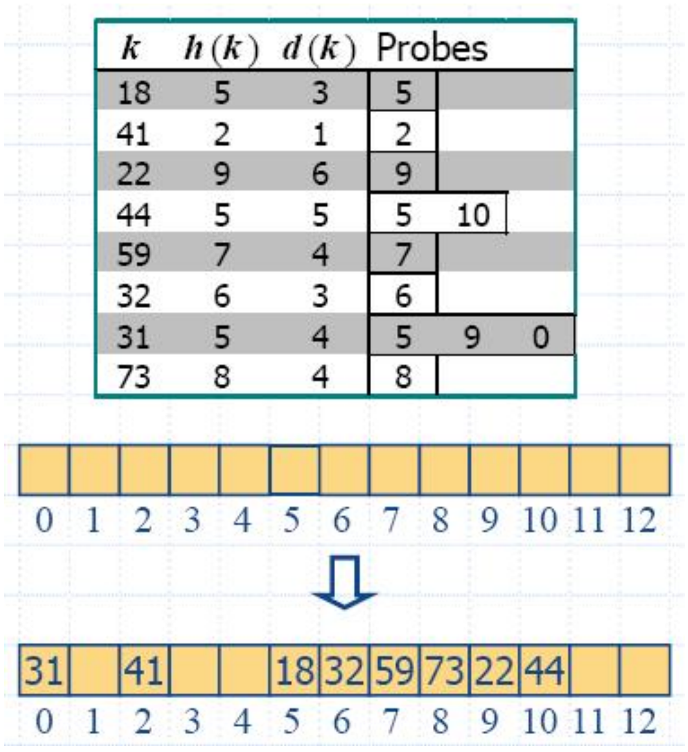
```
while (HT[index] != NULL)
    index= (index+1)%TABLESIZE
/* only get out of this loop
when get to a vacant spot */
```

This method is generally very fast, but has the disadvantage that performance degenerates very rapidly after the load factor (proportion of array elements filled) exceeds around 0.75.



Double Hashing

If a collision occurs, generate a number using the second hash function, and jump ahead this many elements until a free slot is found. To read the data, use hash1 to find the starting location, and then keep jumping forward by however many spaces are indicated by hash2 until the value is found. Like linear probing, this method is very fast or low load factors, but degrades catastrophically for load factors of above around 0.75.



k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

```
jumpnum = hash2(key)
while (HT[index] != NULL)
    index= hash2(index)%TABLESIZE
```