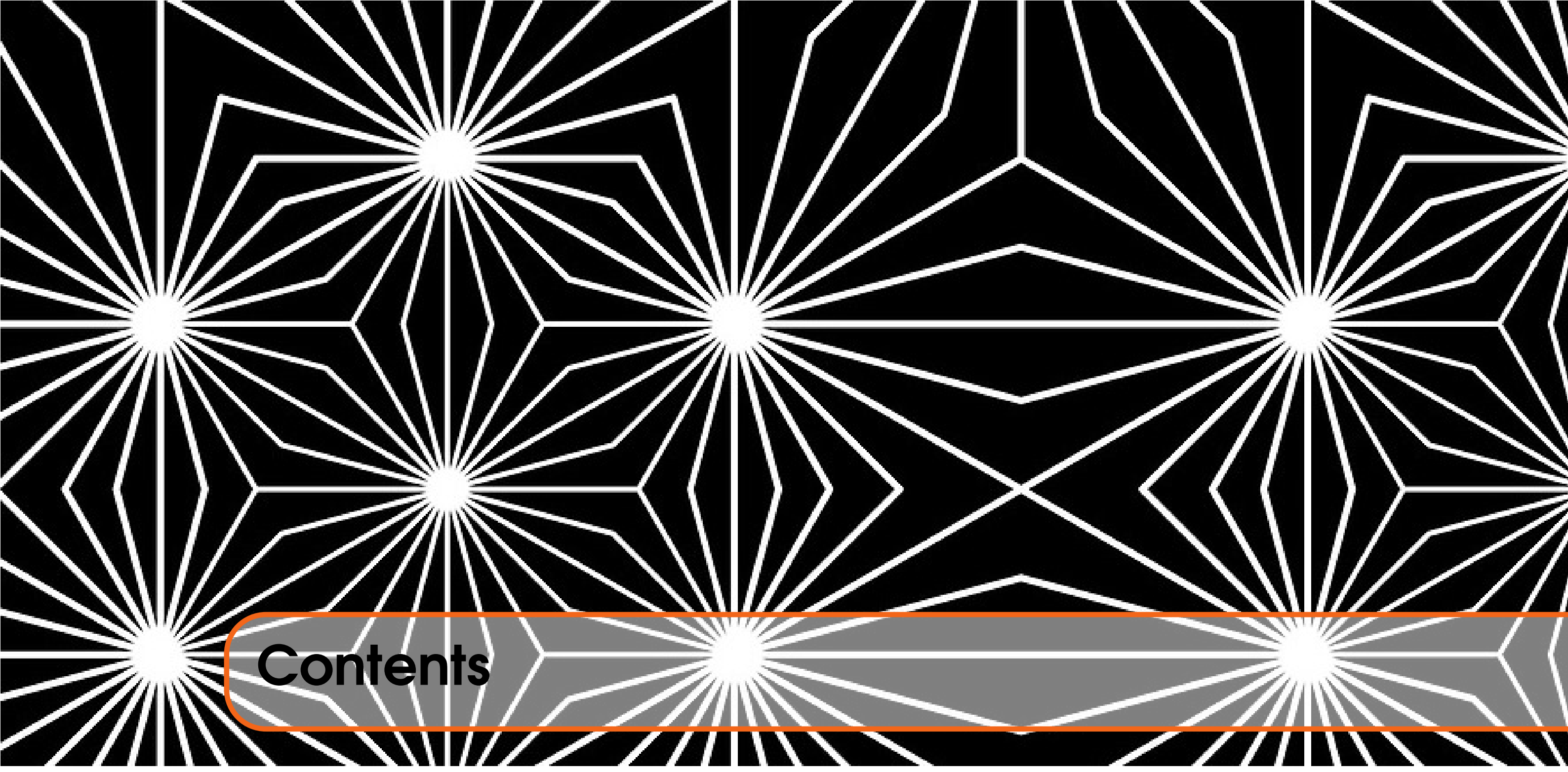
Google Interview preparation Material

The karate guide

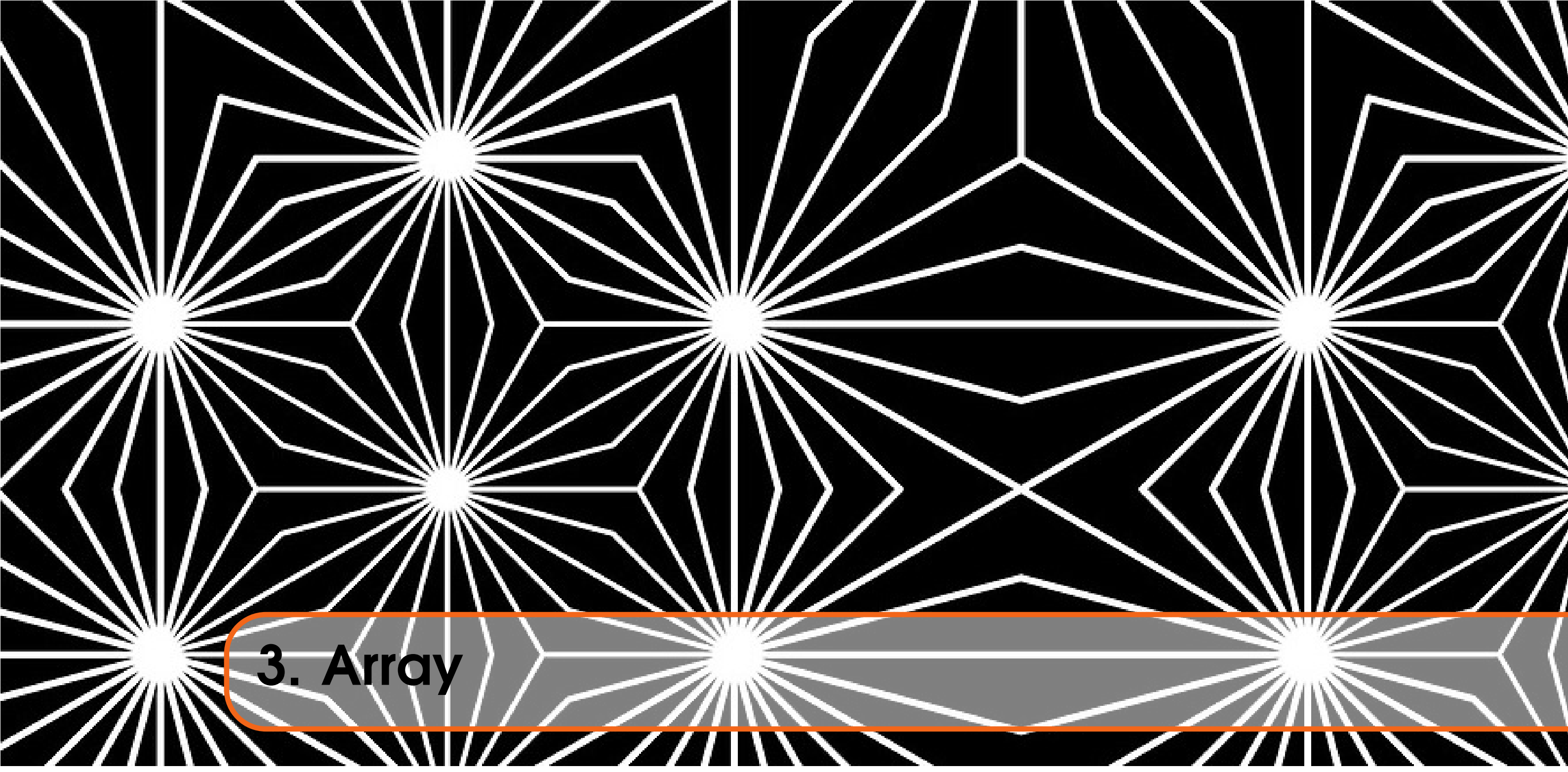
Davide Spataro



Contents

1. Primitive Data Types

2. Composite Data Types



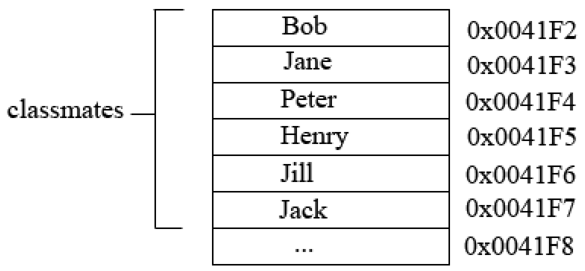
3. Array

Array belongs to the so called *Contiguous* Data Structure in which stored elements (all of the same type) are arranged in a **single** slab of memory. Array's elements can be easily accessed given their *indices* within the array itself the same way, using an analogy, we can locate an house in a specific street given its postcode. Array can also be tought as a collection of variables of the same type.

Major characteristic of arrays are:

Constant time access All the element of the array can be accessed in $\mathcal{O}(1)$. Array can be fully described by its

1. Starting address m_0
2. Size of Stored data type (e.g. 4 bytes for `int` or 1 byte for `char`) S
3. Number of stored elements (i.e. array length) L



The starting address corrенspond to the first element (in C-like languages index 0) element. The second element starts at the address $m_0 + S$, third at $m_0 + S + S$ and so on. so the element at index i has index $s_i = m_0 + iS$.

Only Data Array are space efficient because they do not store any other information but data itself (unlike list for instance where each node of the list stores a pointer to the next element of the list).

Memory locality Array adhere perfectly to the principle of spatial locality[1] in which related

[1]If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will

storage locations are frequently accessed (iterating through an array is a very common idiom in all programming languages). This allows cache to work at his best.

Arrays have fixed size, and this in several application is a major downside (the size of the array is input dependent for instance). C-like languages allow for dynamic memory allocation at runtime (see **??** at page (**??**). This involves the creation of a (different) larger (how larger is an important point) array and the copy of the first one in the beginning of the newly created one. If the program requires this dynamic enlarging often we could end up wasting much time in copying data here and there. The common strategy used for creating a dynamic array is to double the size of the array each time. This mitigate the average number of copies of each element. Let's imagine we start with an array of size one and we try to insert n elements, doubling the size of the array when its capacity its full. The first element is has to be copied when the array expands after the fist, second, fourth, eighth ... $2^i = n$ insertion. It will take $i = log(n)$ doubling size for the array to have size n. Element in the last half of the array will be copied only one (at the last doubling). A quarter of the elements will be copied twic and so on. The total number of copies will be then

$$M = \sum_{i=1}^{\log(n)} \frac{ni}{2^i} = n \sum_{i=1}^{\log(n)} \frac{i}{2^i} = n(\frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{1}{3} \ldots \frac{log(n)}{n}) = 2n$$

So in average each element is copied only two times.

3.1 Problems and Solution

Exercise 3.1 Given an array, A, of N integers, print each element in reverse order as a single line of space-separated integers.

■ Solution 3.1

```c
        int main(){
    int n;
    scanf("%d",&n);
    int *arr = malloc(sizeof(int) * n);
    for(int arr_i = 0; arr_i < n; arr_i++){
       scanf("%d",&arr[arr_i]);
    }
    while(--n >=0){
       printf("%i ",arr[n]);
    }

    return 0;
}
```

Listing 3.1: "C"

Exercise 3.2 Reverse an array in place.

Exercise 3.3 Implement a function to determine if a string has all unique characters. String can only contains latin characters. What happen if you are forced to use no additional data structures?

Exercise 3.4 Implement a function to determine if a string has all unique characters. String can only contains latin characters. What happen if you are forced to use no additional data structures?

Exercise 3.5 Implement a function to determine the number of repeated characters in a string. (e.g. google has 2 repeated characters, g and o).

Exercise 3.6 Given an array of integer of given size containing numbers from 1 to $N \geq 2$. Only one number is missing within the array. Write a function which output is that missing number.

be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access.

Exercise 3.7 Given an array of integer of size N containing numbers from 1 to $N-2$. All numbers appear only once except one which is repeated twice. Write a function which return that number.
Exercise 3.8 Write a function which find the smallest and largest number in an unsorted array of integers.
Exercise 3.9 Sort an array of integer using a $\mathcal{O}(n^2)$ and a $\mathcal{O}(nlog(n))$ algorithm. Both algorithm should sort the array *in-place*.
Exercise 3.10 Given two array of sorted integer, A,B find their intersection i.e. an array C which contains only elements which belong both to A and B. $C = \{c_i \mid c_i \in A;,\ c_i \in B\}$
Exercise 3.11 Given two array of unsorted integer in which only one element is repeted (the rest of them appear only once). Write a function which finds that element in $\mathcal{O}(n)$ time.
Exercise 3.12 Write a function which finds the kth largest element in a unsorted array.
Exercise 3.13 Write a function which finds the kth smallest element in a unsorted array.
Exercise 3.14 Count the number of distinct ((a,b) is not distinct from (b,a)) pairs in an array of integers which sum is equal to a given value K.
Exercise 3.15 Given three sorted array (in non decreasing order) find the intersection between all of their elements.
Exercise 3.16 Given an array, find the first repeated element. i.e. an element e which occurs more than once and which index is the smallest. On the following input $1,2,3,4,5,3,2,2,7$ the function should return two because is the a repeated element with the smallest index (1).
Exercise 3.17 Given an array, return the larger and the 2th-largst element.
Exercise 3.18 Given an array of integer, find the smallest positive integer which cannot be represented as sum of any subset of the array. On input 1, 3, 6, 10, 11, 15 the function should return 2. Note that the array can also contains negative numbers.
Exercise 3.19 Given an array of integer (positive and negative), return an array which elements are rearranged in alternating positive and negative. You are ensured that the number of positives and negatives matches. E.g. on input $(1,2,-6,4,8,-6,-5)$ the functions (only one of the possible valide output) return $(1,-6,2,-6,4,-5,8)$.
Exercise 3.20 Given an array of integer write a function which return true is a non empty subset of its element which sum up to 0. False otherwise.
Exercise 3.21 Given an array A of integer find the length of the longest sequence of consecutive integers. On input $1,56,8,-5,3,7,4,23,2,5)$ the function should return 5 (is the length of the consecutive subsequence $(1,2,3,4,5)$)
Exercise 3.22 Find the minimum in a rotated sorted array. (a rotation of $1,2,3,4,5,6$ could be for instance $4,5,6,1,2,3$). The array does not contains duplicates.

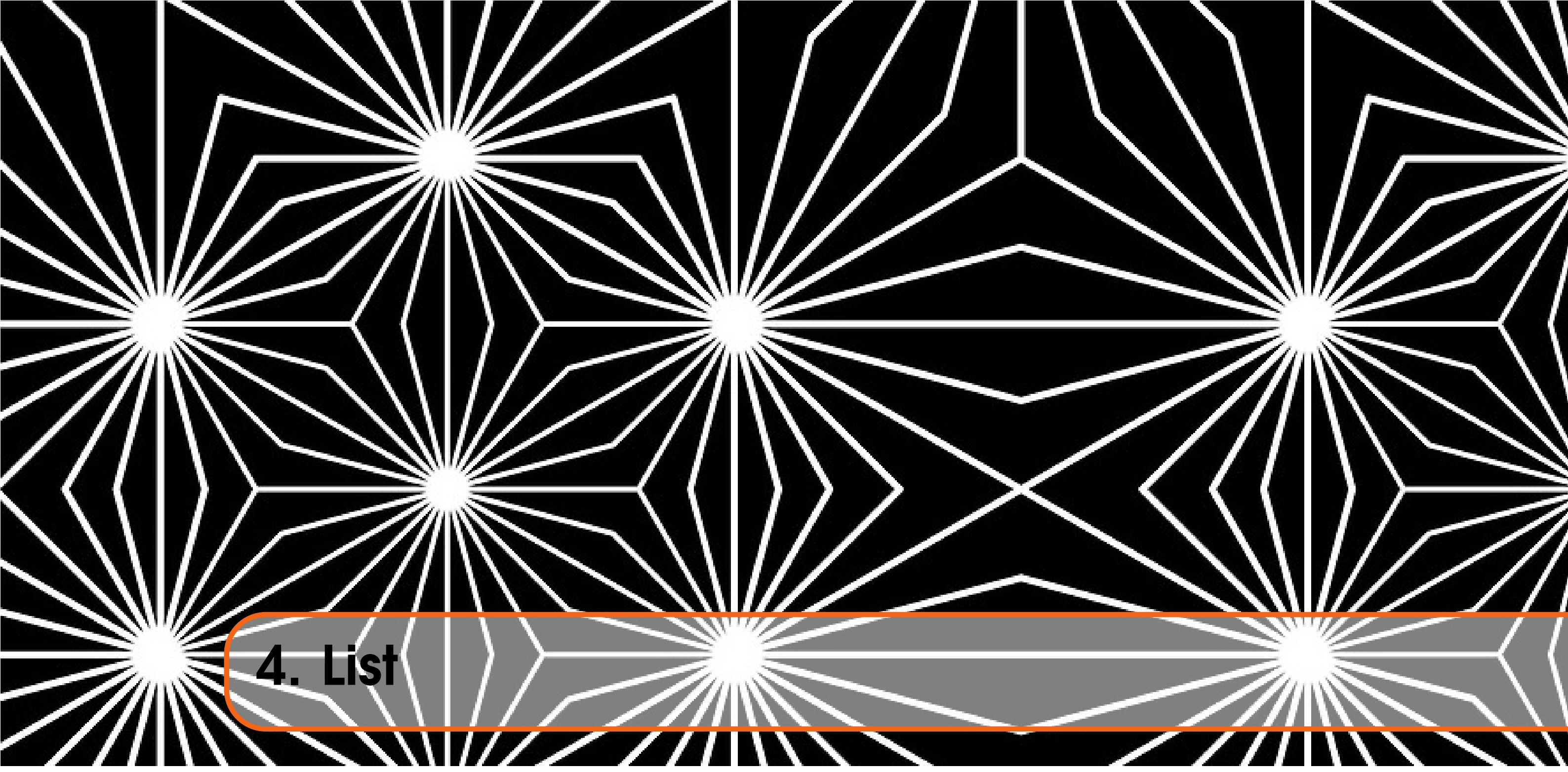
What if we allows for duplicated to be present? How does this changes the overall time complexity?
Exercise 3.23 Given two string, determine which is the minimum number of deletions (a delete operation can be performed on both string) necessary for the two string to be a valid anagram[2] of each other. Strings contains only latin characters with no space. Given *hello* and *belloz* the function should returns 2. (See solution 3.2 at page 11)

```c
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

void removeTrailingNL(char* s){
    int size = strlen(s);
    if(size > 0 && s[size-1] == '\n')
```

[2]A word, phrase, or name formed by rearranging the letters of another, such as spar, formed from rasp.

```c
        s[size-1] = '\0';
}

#define MAX_INPUT_SIZE (10000)
int main() {
    const int alph_size = 'z'-'a'+1;
    int s1_freq[alph_size];
    int s2_freq[alph_size];
    //initialize frequencies arrays
    for(int i = 0 ; i < alph_size ; i++){
        s1_freq[i] = 0;
        s2_freq[i] = 0;
    }
    //read both string
    char s1[10000];
    char s2[10000];
    fgets(s1,MAX_INPUT_SIZE,stdin);
    fgets(s2,MAX_INPUT_SIZE,stdin);
    removeTrailingNL(s1);
    removeTrailingNL(s2);

    int size_s1, size_s2;
    size_s1 = strlen(s1);
    size_s2 = strlen(s2);
    //compute frequencies
    for(int i  = 0 ; i< size_s1 ; i++)
        s1_freq[s1[i]-'a'] = s1_freq[s1[i]-'a'] + 1;
    for(int i  = 0 ; i< size_s2 ; i++)
        s2_freq[s2[i]-'a'] = s2_freq[s2[i]-'a'] + 1;
    /*
  //print frequencies
 for(int i  = 0 ; i< alph_size ; i++){
     printf("%c %i %i\n",'a'+i,s1_freq[i],s2_freq[i]);
 }*/

    //compute minimum delete operation
    int dels = 0;
    for(int i  = 0 ; i< alph_size ; i++)
        dels = dels + (abs(s1_freq[i] - s2_freq[i]));


    //prints output
    printf("%i",dels);

    return 0;
}
```

Listing 3.2: Solution to problem 3.23



4. List

5. Stack

6. Queue

6.1 Double-ended queue - Dequeue

6.2 Priority Queue

Nevertheless, the heap data structure itself has enormous utility. In this section, we present one of the most popular applications of a heap: its use as an efficient priority queue. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A priority queue supports the following operations.

`INSERT(S,x)` inserts an element x into the set S
`{MAX,MIN}(S,x)` return the element of S with largest/smallest key
`EXTRACT-{MAX,MIN}(S,x)` return and removes the element of S with largest/smallest key

One application of priority queues is to schedule jobs on a shared computer. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

A priority queue can also be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. For this application, it is natural to reverse the linear order of the priority queue and support the operations MINIMUM and EXTRACT-MIN instead of MAXIMUM and EXTRACT-MAX. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the priority queue using INSERT.

Not surprisingly, we can use a heap to implement a priority queue. The operation HEAP-MAXIMUM returns the maximum heap element in (1) time by simply returning the value A[1] in the heap. The HEAP-EXTRACT-MAX procedure is similar to the for loop body (lines 3-5) of the

HEAPSORT procedure:

```
Data: Heap H
Result: Extract Max - Max element of H
if $heap\text{-}size(H) < 1$ then
    underflow error
$max \leftarrow H[1]$;
$H[1] \leftarrow H[heap-size(H)]$;
$heap-size(H) \leftarrow heap-size(H)-1$;
return $max$;
```

Algorithm 1: Priority Queue Extract-max pseudocode

The running time of HEAP-EXTRACT-MAX is $\mathcal{O}(logn)$, since it performs only a constant amount of work on top of the $\mathcal{O}(logn)$ time for HEAPIFY.

The HEAP-INSERT procedure inserts a node into heap A. To do so, it first expands the heap by adding a new leaf to the tree. Then, in a manner reminiscent of the insertion loop of INSERTION-SORT (see section **??** at page **??**), it traverses a path from this leaf toward the root to find a proper place for the new element.

```
Data: Heap H, Key key
Result: Insert an element in the Heap H
$heap-size(H) \leftarrow heap-size(H)+1$;
$i \leftarrow heap-size(H)$;
while $i > 1 and H[parent(i)] < key$ do
    $H[i] \leftarrow H[parent(i)]$;
    $i \leftarrow parent(i)$;
end
$H[i] \leftarrow key$
```

Algorithm 2: Priority Queue insert pseudocode

Exercise 6.1 Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.

Exercise 6.2 Give an O(lg n)-time implementation of the procedure HEAP-INCREASE-KEY(A, i, k), which sets A[i] max(A[i],k) and updates the heap structure appropriately.

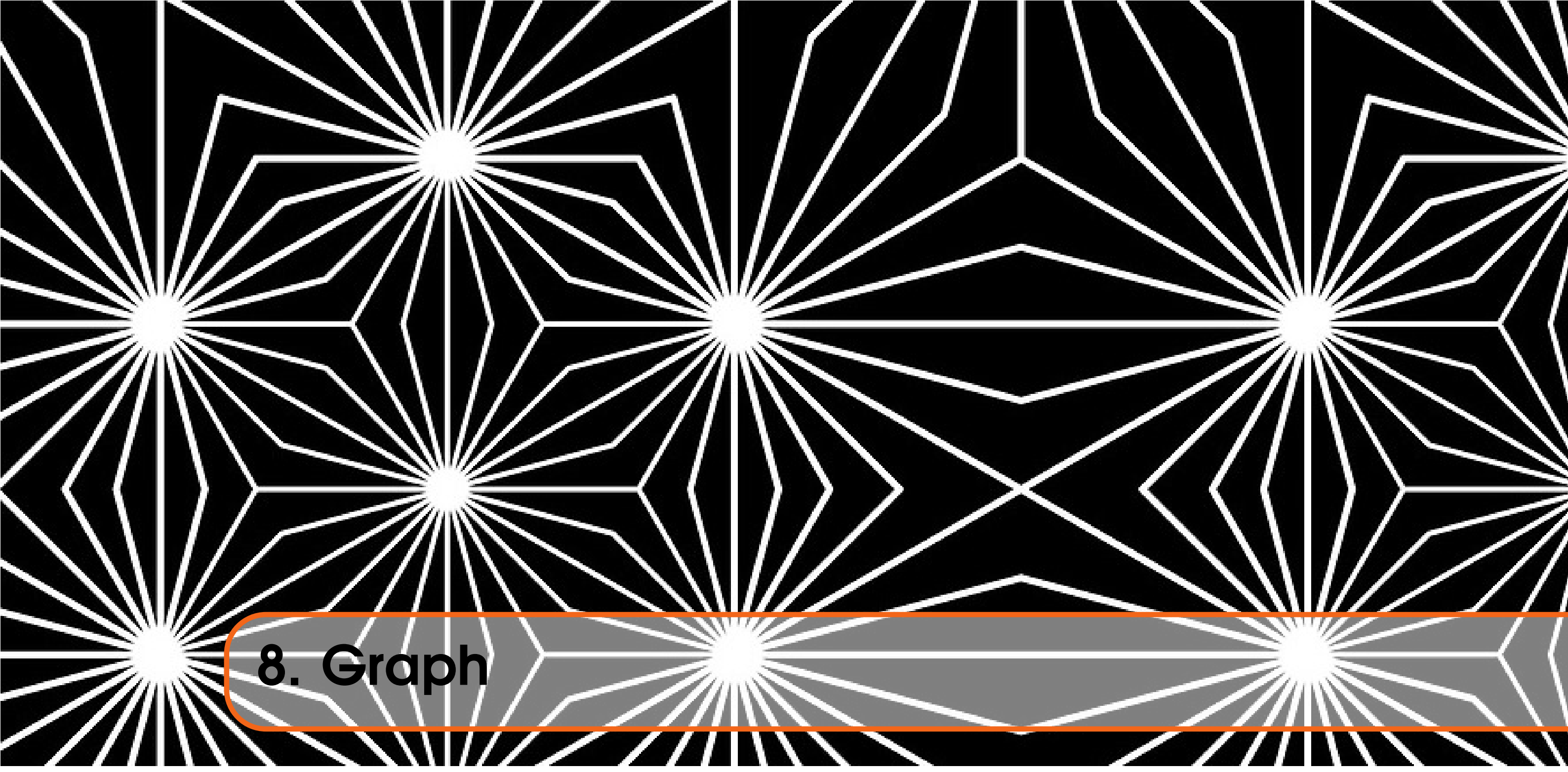
Exercise 6.3 The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in O(lg n) time for an n-element heap.

Exercise 6.4 Give an O(n lg k)-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a heap for k-way merging.)

Exercise 6.5 A d-ary heap is like a binary heap, but instead of 2 children, nodes have d children.

1. How would you represent a d-ary heap in an array?
2. What is the height of a d-ary heap of n elements in terms of n and d?
3. Give an efficient implementation of EXTRACT-MAX. Analyze its running time in terms of d and n.
4. Give an efficient implementation of INSERT. Analyze its running time in terms of d and n.
5. Give an efficient implementation of HEAP-INCREASE-KEY(A, i, k), which sets A[i] max(A[i], k) and updates the heap structure appropriately. Analyze its running time in terms of d and n.

7. Heap



8. Graph

8.1 Maximum Flow Problem

Given a directed graph with $G = (V, E, c)$ where $c(u, v) \in \mathscr{R}, (u, v) \in E$ is a function that specifies the capacity of an edge the goal is to find the maximum flow from two nodes $s \neq t \in V$ called *source* and *sink*. Usually the source has not incoming edges while the sink has not outgoing edges. A flow is a function $f(u, v)$ that specifies the amount of good moving along the (u, v) edge s.t.

- $f(u, v) \leq c(u, v)$ i.e. the flow associated with the edge does not exceed its capacity,
- $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$, $v \neq s, v \neq t$, i.e. for all node v other than the sink and the source the amount of flow entering it is equal to the flow exiting v. Intuitively it means that all the flow that is pushed in the source travels to the sink.

Goal of the maximum flow problem is to find the maximum amount of flow that leave s (or alternatively that reaches t).

8.1.1 Greedy Approach to the Maximum Flow Problem

Let's try to determine the maximum flow in a network via a greedy approach. At this stage is not clear whether such approach would produce an optimal solution. The approach consists in starting with zero flows for all the edges and greedily updating them as we figure out that we can send more

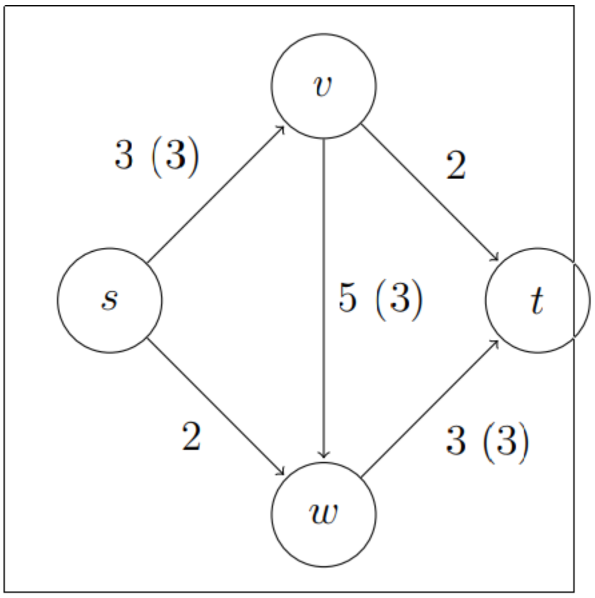


Figure 8.1: An example of how the greedy algorithm return a suboptimal flow.

flow from s to t.

```
Data: Graph G
Result: The maximum flow through the network
foreach $(u,v) \in E$ do
    $f_{(u,v)} \leftarrow 0$;
end
while $P$ path in $G$. do
    $\Delta = \min_{(u,v)\in P}(c(u,v) - f(u,v))$ minimal residual capacity of edges in $P$;
    foreach $(u,v) \in P$ do
        $f_{(u,v)} \leftarrow f_{(u,v)} + \Delta$ add newly discovered flow to all the edges of $P$.
    end
end
```

Algorithm 3: Greedy Maximum Flow approach

Note that 3 has only to search for paths in G taking care of making sure the constraints are satisfied. This is easily done with BFS or DFS in linear time (in the number of edges). Note also that there may be many of such paths, and the algorithm chooses one aritrarly. **This algorithm is not optimal, i.e. find a suboptimal solution to the problem.** as can be seen in Figure 8.1.1. Regarding the Figure 8.1.1, imagine that the first path that the algorithm considers is the zig-zag one: s, v, w, t. Δ would be $\min 3, 5, 3 = 3$. This means that at the subsequent iteration the graph would not contain any path from s to t at all, giving a result of 3, while there is a flow of 5 if we allow to route a flow of 2 via s, v, t and a flow of 2 via s, W, t and a flow of 1 via s, v, w, t.

Luckily the if the same approach is applied to a different kind of graph derived from the original one, the algorithm leads to the optimal solution. This derived graph is called: *residual graph*.

Residual Graph

The idea is to be able to perform kind of *undo* operation in such cases where the greedy algorithm get stuck as seen in the previous example. The idea is to augment the original graph by having a pair of edges for each of the original edges. One edge specifies the flow $f(u,v)$ and the other one residual capacity i.e. $c(u,v) - f(u,v)$.

8.2 Exercices

Exercise 8.1 The incidence matrix of a directed graph $G = (V,E)$ with s.t. E is a not reflexive relation is a $V \times E$ matrix $B = b_{ij}$ such that:

$$b_{ij} = \begin{cases} -1 & \text{if j leaves i} \\ 1 & \text{if j enters i} \\ 0 & \text{otherwise} \end{cases}$$

What the entries of BB^T represents?

■ **Solution 8.1**

Exercise 8.2 *Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?*

■ **Solution 8.2**

- Out-degree is easy since each node stores the adjacency information as out edges. So out-degree is simply the length of the list which is usually stored as list's attribute hence accessible in $O(1)$. If it is the case the time complexity is $O(|V|)$ otherwise we need to inspect all the edges yelding to a $O(|E|)$.
- In-degree for node i requires to collect adjacency information from all the other nodes $j \neq i$. Complexity is $O(|E|)$.

Exercise 8.3 *Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.*

■ **Solution 8.3** **Adjancency Matrix**

	1	**2**	**3**	**4**	**5**	**6**	7
1	*0*	*1*	*1*	*0*	*0*	*0*	*0*
2	*0*	*0*	*0*	*1*	*1*	*0*	*0*
3	*0*	*0*	*0*	*0*	*0*	*1*	*1*
4	*0*	*0*	*0*	*0*	*0*	*0*	*0*
5	*0*	*0*	*0*	*0*	*0*	*0*	*0*
6	*0*	*0*	*0*	*0*	*0*	*0*	*0*
7	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Adjancency list

```
1:  2 -> 3 -> NIL
2:  3 -> 4 -> NIL
3:  5 -> 6 -> NIL
4: NIL
5: NIL
6: NIL
7: NIL
```

Exercise 8.4 The transpose of a directed graph $G = (V,E)$ is the graph $G = (V,E^T)$, where $E^T = \{(v,u) \in V \times V : (u,v) \in E\}$. Thus G^T is G with all edges reversed. Describe an efficient algorithm for computing G^T for both list-matrix adjacency graph representation. What is the running time complexity of your solutions?

■ **Solution 8.4** **Adjacency List**

The previous algorithms runs in $O(|E|)$. Is thus linear in the number of edges.

```
Data: Graph $G = (V,E)$
Result: graph $G^T = (V,E^T)$
$E^T[v.size()] \leftarrow$ initialize adjacency lists;
for $u \in V$ do
    for $(u,v) \in E[u]$ do
        $E^T[v] \leftarrow (v,u) : E^T[v]$;
    end
end
return $(V,E^T)$;
```

Adjacency Matrix
Adjacency matrix for E^T is simply the transpose of E. It runs in linear time in the size of matrix which for dense graph is linear in the number of edges in G, but for sparse graph is quadratic (yelding so to a quadratic time algorithm).

Exercise 8.5 Given an adjacency-list representation of a multigraph $G = (V,E)$, describe an $O(V+E)$ time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G = (V,E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

■ **Solution 8.5** Solution is easy if we use a visiting algorithm which takes care to avoid cycles and visit every nodes of each connected component exaclty once (this implies that whenever two nodes (u,v) are connected by more than once edge we process only one of them, otherwise we would visit v twice, which is contradicts with the the fact that we visit a node only once). This runs in $O(V+E)$.

```
Data: Graph $G = (V,E)$
Result: graph $G^T = (V,E')$
$E^T[v.size()] \leftarrow$ initialize adjacency lists;
$visited[v.size()] \leftarrow FALSE$;
for $n \in V$ do
    if $visited[n] == FALSE$ then
        $Q.queue(s)$;
        while $!Q.empty()$ do
            $u \leftarrow Q.dequeue()$;
            $visited[u] = TRUE$;
            for $v \in E[u]$ do
                if $visited[v] == FALSE$ then
                    $E'[u] \leftarrow v : E'[u]$;
                    $E'[v] \leftarrow u : E'[v]$;
                end
            end
        end
    end
end
return $(V,E')$;
```

Exercise 8.6 Most graph algorithms that take an adjacency-matrix representation as input re- quire time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a universal sink a vertex with $|V|-1$ in-degree and out-degree 0 in $O(V)$ using adjacency matrix.

■ **Solution 8.6** We compute the sum of all the column of the matrix. Whenever we found a 1 set the corrensponding row as a not suitable candidate as sink element because its out-degree is at least one. We then check is there is some column sum which sums up to $|V|-1$ and whose boolean flag is not set to false.

8.3 Visiting Algorithm

8.3.1 Breadth-first

Given a graph $G=(V,E)$ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It has the convenient property that in unweighted graph it computes the minimum distance from s to all the node reachable from s itself producing the so called breadth-first tree with root s. It takes its name from the fact that is first expand the frontier of the currently visited node. It first examine all the node at ditance k from the root before to visit those at distance $k+1$. Each node is labeled as undiscovered, discovered or processed, reflecting the different states in which a node can be during the visit.

The breadth tree is initially initialized with the root s. Then whenever the search encounter a undiscovered node v it is inserted among those to be visited and labeled as discovered and the edge that lead from the examined node to v is added to the breadth tree. Since we visit every node only once and for each node we examine its adjacency list in its full length the complexity is is $O(V+E)$

8.3.2 Why Breadth-First computes the shortest path

Let $\delta(s,u)$ be the shortest path (or minimum distance) between two vertices s,v[1]. If a path does not exists then $\delta(s,u)=\infty$. Note also that $\delta(s,s)=0$.

Theorem 8.3.1 Tringular inequality The triangular inqeuality can be expressed using the following formula:

$$\forall x,y,w \mid \delta(x,w) \leq \delta(x,y)+\delta(y,w)$$

It says that the shortest path between x,w cannot is always equal or shorter than the path from from x leads to w passing from an intermediate node y (see image 8.3.2)[a].

In other words given a vertex s and an edge (u,v) then the following holds:

$$\delta(s,v) \leq \delta(s,u)+1$$

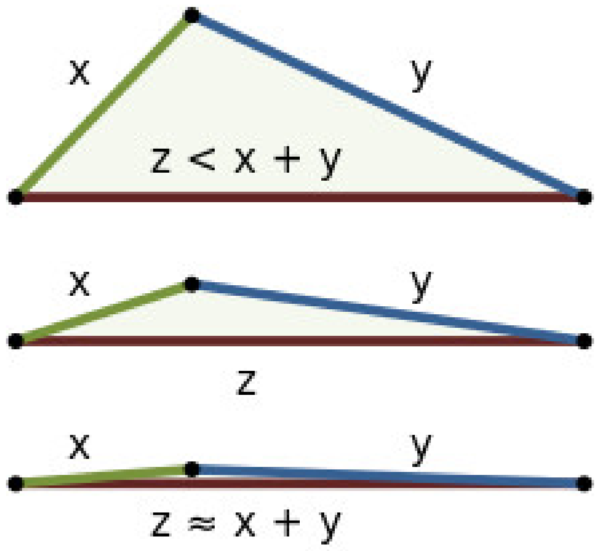
If u is reachable from s so is v and in particular that $\delta(s,v)$ cannot be longer than going from s to u first and then traverse the edge (u,v) leading to the following walk:

$$s \rightarrow v_1 \rightarrow \dots v_{\delta(s,v)-1} \rightarrow u \rightarrow v$$

[a] Is called triangular because is can be easily pictured using a triangle. The distance between vertices on the hypotenuse cannot be longer than the length of the sum of both catheti

We want to show that BFS computes the shortest path $\delta(s,v)$ correctly. We will do it showing first that the distance computed by the BFS procedure bound $\delta(s,v)$ from above.

[1] Note that for undirected graphs this distance is symmetric i.e. $\delta(s,u)=\delta(u,s)$



Theorem 8.3.2 Suppose we run BFS on a graph G from node s then $v.d \geq \delta(s,v)$. The proof is by induction on the number of `ENQUEUE` operations. The base case is when only s is in the queue so $s.d = 0 = \delta(s,s)$ and $v.d = \infty \geq \delta(s,v)$. Suppose that the hypotesis holds for a number of enqueue operation and in particular that an undiscovered vertex is enqueue from a parent u. The inductive hypothesis implies that $u.d \geq \delta(s,u)$. BFS computes the path from s to v composing the path from s to u and adding an edge from u to v. this means that: $v.d = u.d + 1$ From the theorem 8.3.1 follows that $v.d = u.d + 1 \geq \delta(s,u) + 1 \geq \delta(s,v)$.

We have bounded BFS path length from s from above. BFS computes a path which is clearly always less or equal shorter than the shortest possible one. We move on proving that the path it computes are actually the shortest possible so $v.d = \delta(s,v)$. We now shows that BFS manage in its queue at most two distinct distance values.

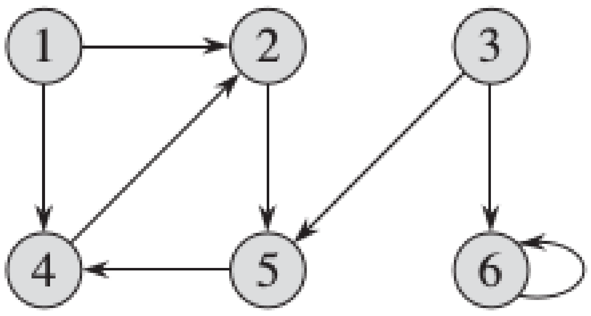
Theorem 8.3.3 **BFS computes the shortest path from s to all reachable nodes**
Suppose that for some node (pick up the closest to s) $v.d \neq \delta(s,v)$, then follows that $v.d \geq \delta(s,v) \wedge v.d \neq \delta(s,v) \Rightarrow v.d > \delta(s,v)$. Let (u,v) an edge then it follows that $\delta(s,v) = \delta(s,u) + 1 = u.d + 1$ since v is the first node for which the inequality hold than the BFS path from s to u is the shortest possible[a]. Putting all together we obtain[b]

$$v.d > u.d + 1$$

Consider now the status of v at the time u is explored by BFS procedure. It can be one of the following three cases:

1. v is UNDISCOVERED: so its distance from s will be $u.d + 1 \Rightarrow$ **(CONTRADICTION)**
2. v was already processed (earlier): $v.d \leq u.d \Rightarrow$ **(CONTRADITION)**
3. v is DISCOVERED but not PROCESSED: This means that v was discovered when processing a vertex w before u was examined. This also implies that w was processed before u and hence queued before u. This implies that $w.d \leq u.d \Rightarrow w.d + 1 \leq u.d + 1$. Now, v was discovered by w so its distance from s is $w.d + 1$. This allows us to conclude that $v.d \leq u.d + 1 \Rightarrow$**(CONTRADICTION)**.

[a] $\delta(s,u) = u.d$
[b] $\delta(s,v) = \delta(s,u) + 1 = u.d + 1$



BFS build the so called *Breadth tree* which consist of all the simple (and shortest) paths from s to any reachable (from s) vertices [2]. Note that this tree is not unique, and depends on the order in which nodes in the adjacency lists are examined. Node can be discovered using different edges.

8.3.3 Exercices

Exercise 8.7 There are two types of professional wrestlers: babyfaces (good guys) and heels (bad guys). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O.n C r/-time algo- rithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

■ **Solution 8.7** This is a two-color graph coloring problem which is easily solvable in linear time (we are also asking if the graph is bipartite). We create a node for each professional wrestler and connect them using the relation coded into the r pairs. We then start coloring the graph from a random node and at each level we switch color. If we encounter a node which was previously colored with a different color[3] the such graph is not bipartite and so is not possible to divide *babyfaces* from *heels*. If we consume all the nodes, then the output graph is a proper bipartition.

Exercise 8.8 *Show the d and π values that result from running breadth-first search on graph pictured in figure 8.8, using vertex 3 as the source.*

■ **Solution 8.8**

$$d(3) = 0, d(1) = \infty, d(5) = d(6) = 1, d(4) = 2, d(2) = 3$$

.

$$d(3) = NIL, \pi(1) = NIL, \pi(5) = \pi(6) = 3, \pi(4) = 5, \pi(2) = 4$$

.

Exercise 8.9 *Show the d and π values that result from running breadth-first search on graph pictured in figure 8.9, using vertex 3 as the source.*

■ **Solution 8.9**

$$d(3) = 0, d(1) = \infty, d(5) = d(6) = 1, d(4) = 2, d(2) = 3$$

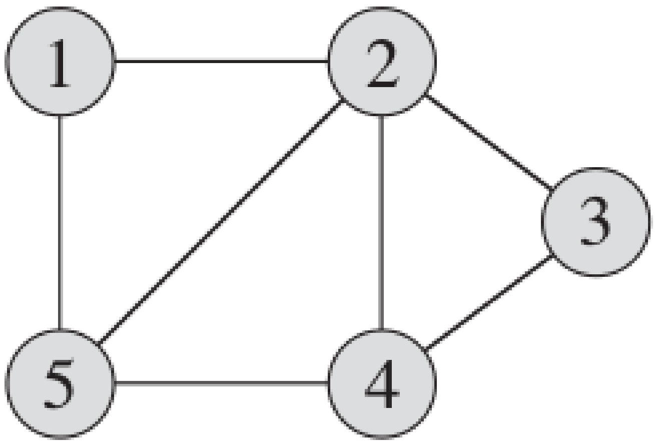
.

$$d(3) = NIL, \pi(1) = NIL, \pi(5) = \pi(6) = 3, \pi(4) = 5, \pi(2) = 4$$

.

[2]Those vertices which have a not null parent

[3]This means that two vertex on a edge that were preivously coloured with different colors are now of the same color.



8.3.4 Depth-first

Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search *backtracks* to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. As for breadth-first if any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source until all the vertices are discovered. As in breadth-first search, depth-first search colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. Depth-first also timestamps each node at the time of discovery and when it finishes to process it (consumes its entire adjacency list) in, respectively $v.d$ and $v.f$ s,t, $v.d < v.f$.

```
Data: Graph $G = (V,E)$
for $v \in V$ do
    $status[v] = UNDISCOVERED$;
    $parent[v] = NIL$;
end
for $v \in V$ do
    if $status[v] == UNDISCOVERED$ then
        $DFS-helper(G,v)$;
    end
end
```

Algorithm 4: DFS procedure

Every time in the main procedure DFS-helper is called, v becames a root of a depth visit tree. Note that the resulting visiting forest is not unique and may depend in the order in which the vertices are processed. This is not usually a problem in practice since we can equivalently use any of the possible ouput forest.

What is the running time of DFS? The main procedure loops over all the vertices and call DFS-helper. DFS-helper is executed only once per vertex (only when their status is undiscovered) and have it loops over all the edges in $ADJ[v]$. The sum of all edges is clearly $|E|$, hence the overall complexity is $\Theta(V+E)$

```
Data: Graph $G = (V, E)$, start node $s$
$time = time + 1$;
$s.d = time$ ;                                   // discovery time for $s$
for $v \in ADJ[V]$ do
    if $status[v] == UNDISCOVERED$ then
        $parent[v] = s$;
        $DFS - helper(G, v)$;
    end
end
$status[s] = PROCESSED$;
$time = time + 1$;
$s.f = time$;
```

Algorithm 5: DFS-helper procedure. Start the DFS from a node s

Analysis and properties of DFS

DFS yelds to imponrtant and valuable information about the structure of the graph. The prodecessor subgraph $G_\pi = (V, E_\pi)$ subgraph is a forest of trees and reflect the recursive structure of the calls to DFS-helper. Another important property is that dicovery and finish time have the so called parenthesis structure, in the sense that the history of discovery and finish time of the vertices make up a well formed nested expression of parenthesis.

Theorem 8.3.4 **Parenthesis theorem for DFS** In any DFS visit for any pair of vertices u and v one and only one of the following property holds (see image 8.3.4). Note that $\forall\ v \in V\ v.d < v.f$

The intervals $(v.d, v.f)$ and $(u.d, u.f)$ are completly disjoint i.e. $v.f < u.d$ or $u.f < v.d$. It means that neither v or v are descendant of the other in the DFS forest. They are discovered in different recursive branches of the DFS-helper procedure.

2. The u's interval is completly overlaps with v's i.e. $v.d < u.d$ and $v.f > u.f$. It means that u is direct descendent of v.
3. The v's interval is completly overlaps with u's i.e. $u.d < v.d$ and $u.f > v.d$. It means that v is direct descendent of u.

8.3.5 Topological Sort

Topological sort is an operation on directed acyclic graphs (also called DAGs). It orders vertices such that all directed edges goes from left to right or right to left (depending on the choosen ordering). Such ordering exists only on direct graphs (because each undirected edge is a cycle of length 2) and on acyclic graphs (because in a cycle there is no way to go in a specified direction withouth going back to the starting point, hence for two nodes i, j both i after j and j after i hold). Each DAG has at least one topological sort. DAG are the natural representation of a number of important problems, such job scheduling for instance where each job is a vertex and edges represent dependencies between them (and edge (i, j) means that job i need to be performed before job j). Topological sortin can be performed using DFS and in particular a graphs ia a DFs if and only if no back edges are encountered during the visit.

In particular when examining an edge (x, y) from x, y can be in one of the following states:

undiscovered : y appers then after x in the topological sort

discovered but no processed : it means that it is a back edge (a cycle exists in the graph). The graph is not a DAG, so no topological sort exists for it.

processed : y was discovered and processed before x, so it appears before in the topological sort

Exercise 8.10 *Prove by induction that exists only one path between each vertices of a tree*

■ **Solution 8.10** We will use induction on the level tree level. We can think about two distinc base cases at least. Tree of height 1 and 2. Tree with height 1 have only one node so the property holds trivially. Node with height 2 have one $N_c + 1$ nodes where N_c is the number of child of the root r. Each child is connected to the root by a unique edge so there exists a unique path between any child and the root. Children are not connected each other so any path is forced to contain r. So the only legal path between childre i and j , $i \neq j$ is of the following form $i \rightarrow p \rightarrow j$.

Suppose now that this property holds for tree with height up to n and see if the property holds for trees with height $n+1$.The nodes at level $n+1$ does not add any edges at levels $n' < n+1$ so this does not add any path between any nodes at upper levels. We will shows that exists a unique path between any node at level n' and $n+1$ and between leafs in two distinc steps.

- The path between the leaf i with parent $p(i)$and a non leaf o is forced to contain the unique edge connecting i and $p(i)$. Since there exists a unique path between o and $p(i)$ there is only one way to construct a path between o and i and is the following: $o \rightarrow p(i) \rightarrow i$
- The path between two leafs i, j with parents $p(i), p(j)$ respectively is forces to contains edges $(i, p(i)), (j, p(j))$ since those are the only edges connecting the them to the upper levels. Let c the first common ancestor of i, j which is clearly at level $n' < n+1$. There is unique path between c and both $p(i)$ and $p(j)$ so the only possible path is the following: $i \rightarrow p(i) \rightarrow c \rightarrow p(j) \rightarrow j$.

Exercise 8.11 *Prove that in BFS every edges is either a tree edge or a cross edge. Shows that in cross edges (x, y) x is neither an ancestor nor a descendant of y (not immediate).*

■ **Solution 8.11** Let's begin saying that edge (x, y) is classified as tree or cross depending on the status the endpoint y and that we first examine all edges going out from x before examining another node. All nodes from level l are fully examined before moving to nodes at deeper levels. If y is in an undiscovered status it means that no other node $z \neq x \neq y$ have been examined such that an edge (z, y) exists so far. It it then labeled as tree edge (there is no other node at previous level which connects to y). Suppose that y has status *discovered*. This means that there exists already an edge with endpoint y in the BFS tree and the edge is labeled as cross. Each node can be only in one of those two state hence each edge is labeled either as cross or tree.In other words another node from the same level has discovered y first. In a cross edge (x, y) x cannot be a non immediate ancestor of y because if it was the case we would have used that edge as tree edge when visiting x. The same hold for an non immediate descendant because the graph is undirected and we would have visited y before x and used (y, x) as tree edge.

Notice that BFS consider edges from level l to level $l+1$. Being a non immediate ancestor or descendant of a node implies to be processed at previous levels.

Exercise 8.12 *Give a linear algorithm to compute the chromatic number of graphs where each vertex has degree at most 2. Must such graph be dipartite?*[4]

■ **Solution 8.12** We first derive un upper bound for the chromatic number, i.e. which is the minimum number $U_\gamma(G)$ for a graph G s.t. $\gamma(G) \geq U_\gamma(G)$. Let's first say that each connected component has a chromatic number which is independent from the others and the graph chromatic number would be the maximum between all the component's chromatic numbers. Graphs of this kind are very similar to cycle graphs in which each connected component have at most only two nodes with degree one. Cycle graph have chromatic number 2 or tree depending on the number of vertices (2 if even, 3 if odd). When a connected component has two nodes with degree one, then its chromatic number is always 2 (is like a list in which head and tail have degree 1). A simple algorithm that work in linear time and constant space is the folloowing. If the number of nodes

[4]The chromatic number of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color (Skiena 1990, p. 210), i.e., the smallest value of k possible to obtain a k-coloring. Minimal colorings and chromatic numbers for a sample of graphs are illustrated above

is even return two. Otherwise use DFS from a node s coloured in white. Each time a node is discovered we color it with the opposite color of its parent. Each time we color a vertex we check for any conflict in color. If we find a conflict we simply pick up a new color to be used for the current vertex. Note that there will be only one place for conflict so no more than one node will need a "spare" color.

Note that there is a theorem that generalize this result. The Brooks's theorem which state that for any graph with degree at most Δ its chromatic number is Δ. Two exception to this rule are the complete and cycle graph with an odd number of vertices (infact in this exercice for cycle graph we require 3 colors).

Exercise 8.13 *In breadth-first and depth-first search, an undiscovered node is marked discov- ered when it is first encountered, and marked processed when it has been completely searched. At any given moment, several nodes might be simultaneously in the discovered state.*

- *Describe a graph on n vertices and a particular starting vertex v such that $\Theta(n)$ nodes are simultaneously in the discovered state during a breadth-first search starting from v.*
- *Describe a graph on n vertices and a particular starting vertex v such that $\Theta(n)$ nodes are simultaneously in the discovered state during a depth-first search starting from v*
- *Describe a graph on n vertices and a particular starting vertex v such that at some point $\Theta(n)$ nodes remain undiscovered, while $\Theta(n)$ nodes have been processed during a depth-first search starting from v. (Note, there may also be discovered nodes.)*

■ Solution 8.13
- The Star Graph (see figure 8.13) S_k using as starting node the central one (the unique with degree $k-1$), or the complete graph K_k (see figure **??**) using any node as starting point.
- The cycle graph C_k (see figure **??**) using any node as starting point.
- A graph with n nodes s.t. two half of the nodes are organized as a list of nodes $\frac{n}{2}$, and each their heads share a node (which is used as starting point). DFS will eventually process one branch leaving the other one undiscovered. Since each branch has $\frac{n}{2} = \Theta(n)$ nodes the constraints are satisfied.

Exercise 8.14 Suppose an arithmetic expression is given as a DAG (directed acyclic graph) with common subexpressions removed. Each leaf is an integer and each internal node is one of the standard arithmetical operations $(+, -, \times, \div)$. For example the expression $2+3\times 4+(3\times 4)\div 5$ is represented by the DAG in Figure **??**. Give an algorithm that work in $O(n+m)$ where n and m are the number of vertices and edges respectively.

■ Solution 8.14 This problem is solvable using a DFS and dynamic programming.

```
if $isleaf(n)$ then
    return $n.value$
end
if $cache(n)$ then
    return $cache(n)$
end
$res \leftarrow neutralElement(n.operation)$;
for $v \in n.adj$ do
    $res = n.operator(eval(v))$;
end
$cache(n) \leftarrow res$;
return $res$;
```

Algorithm 6: DAG Expression evaluation

Exercise 8.15 Skiena 5-12

■ **Solution 8.15**

8.3.6 Network Flow Problems

When graph are interpreted as network we can use them to answer questions regarding flow of some ìkind of material or commodity between two nodes using edges are sort of *pipes*. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The *flow* of the material at any point in the system is intuitively the rate at which the material moves. Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks. Each edge has a capacity at which the material can flow through it (as $200l$ per hours). Vertices represents junctions between conduits where the flow is not collected or stopped or slowed down. It is called flow conservation (similar to Kirchhoff law). One famous example of network problem is the maximum flow which aim to compute the max rate at which a flow can flow from a source to a destination (called sink) node.

A flow network is a directed **connected**[5] graph in which each edge has a **positive** capacity, and self-loops exists.

Let $G = (V, E)$ be a network flow with capacity function c (assign a capacity to each arch), s and t *source* and *sink* nodes respectively.

Capacity Constraint For all u and v $0 \leq f(u,v) \leq c(u,v)$. This property states that the flow flowing through edge (u,v,) cannot be larger than its capacity.

Flow Conservation For all $u \in V - s,t$ (all nodes except sink and source)

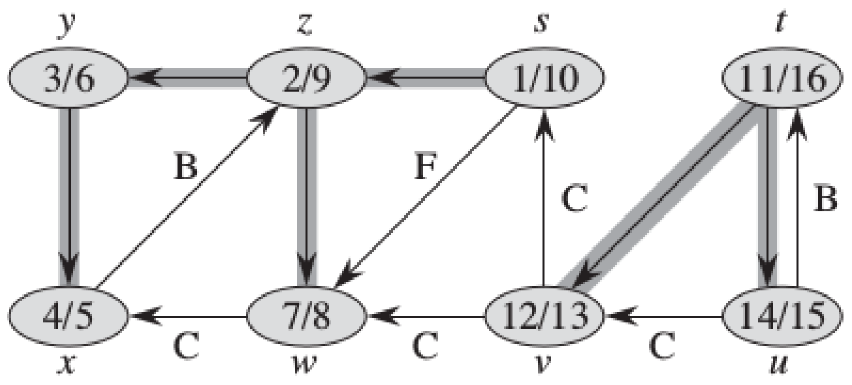
$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

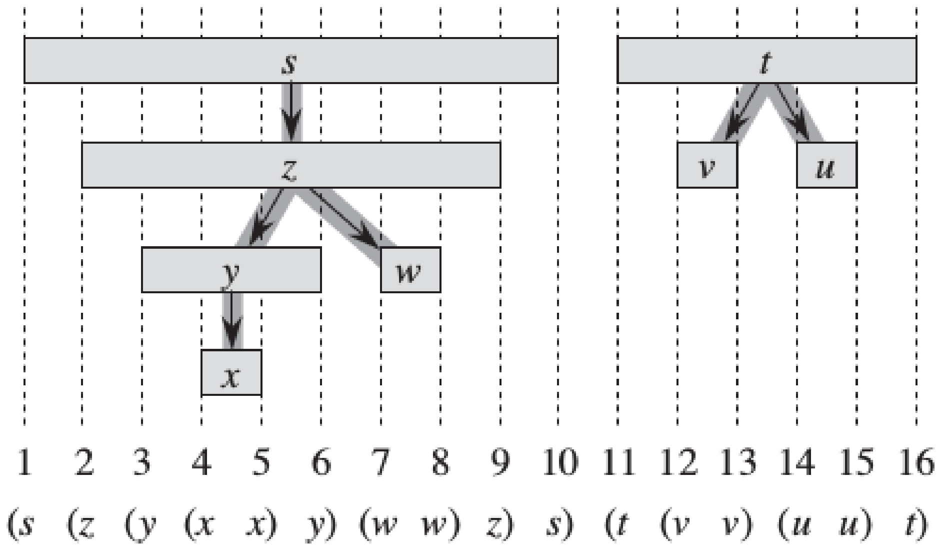
This property ensure that all the flow entering a node inside the network (not a sink or a source) goes out in its entirety (no flow is retained inside the node).

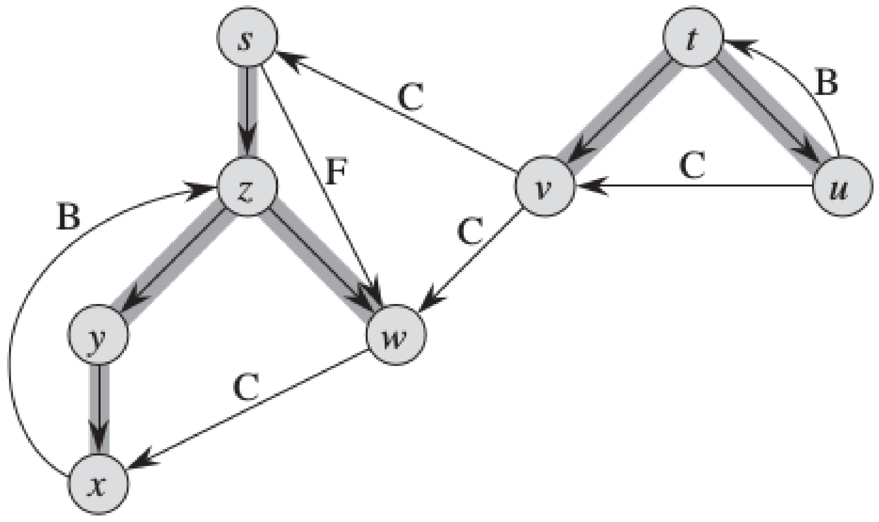
Total flow of the network is defined as difference between inflow and outflow from the source node.

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

[5]for each vertex v a path from the source to v exists. It means that each node has at least one entering arch. This implies that $|E| \geq |V| - 1$


y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)


s
t
C
B
z
F
v
C
u
B
C
y
w
C
x

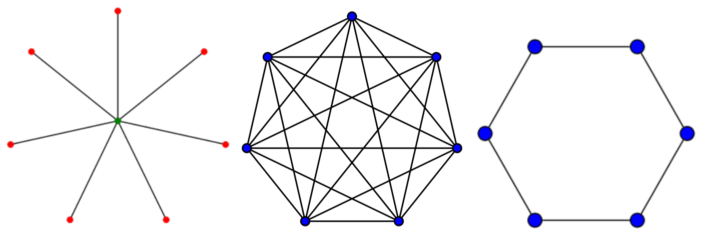
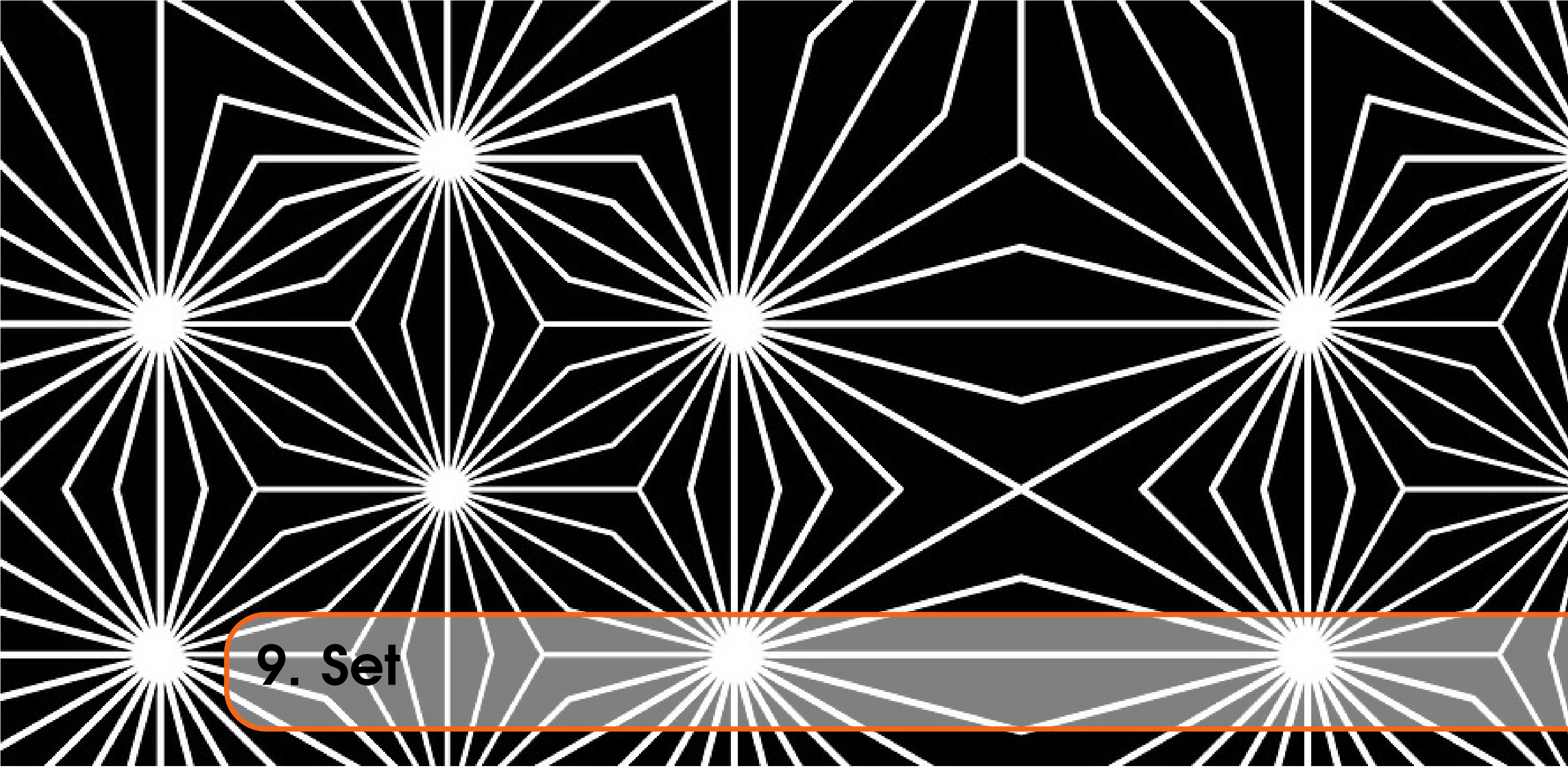


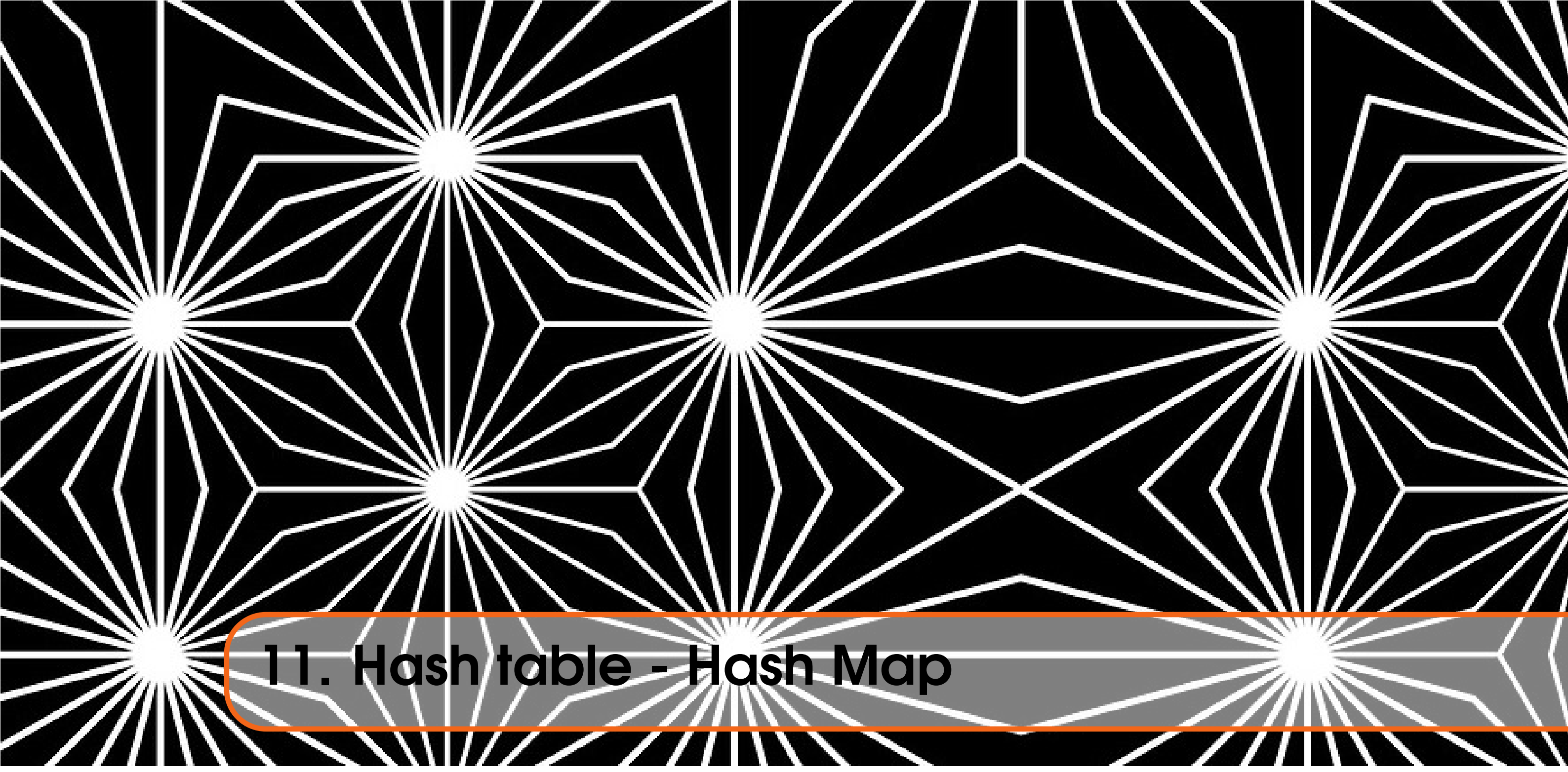
Figure 8.2: Star complete and a cycle graph examples.



9. Set

9.1 MultiSet

10. Associative Array - Dictionary



11. Hash table - Hash Map

Hash tables generalise the simple notion of direct addressing making effective uise of the ability to examine an arbtrary element of an array in $O(1)$ which we know is not affordable when the key universe is big, because it requires space which is proportional to the size of the universe.

11.0.1 Direct Addressing Table

Suppose for instance that an application needs a synamic set in which each elemtn has a keu drqwn from the universe $U0\{0,1,\dots m-1\}$ where m is not too large and no towo element share the same key (indiex of the direct addressing table). A direct addressing table is a indexable structure of size m where each key of the universe has a one to one correspondence to one slot of the table. Dictionary operations can be performed as follows:

```
Procedure SEARCH (T,k)
 |  $return T[k]$
Procedure INSERT (T,x)
 |  $T[tokey(x)] = x$
Procedure DELETE (T,x)
 |  $T[tokey(x)] = NIL$
```

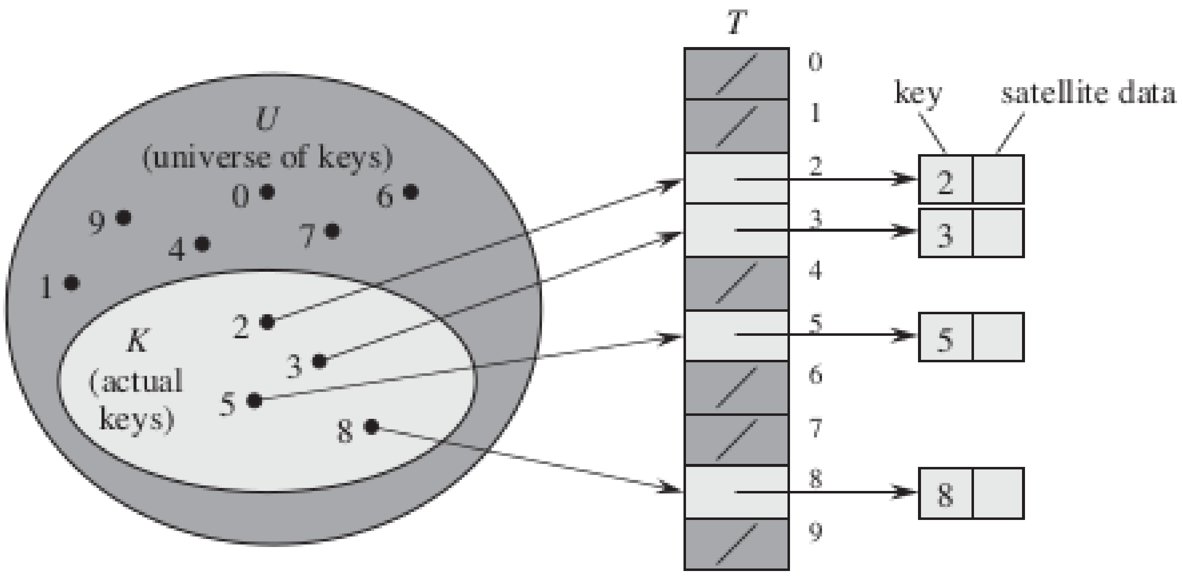
Algorithm 7: DIRECT ADDRESSING OPERATIONS

where *tokey* is a function which takes an object x and maps it to an integer in the interval $0 \dots m-1$. Such function is called $hashing function$.

Exercise 11.1 *Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?.*

■ **Solution 11.1** The direct addressing table is not sorted so binary search cannot be used. Only option is a linear scan.

Hash tables use addressing with indices as basic idea, but overcome the problem of using large space by using space which is proportional to the number of keys actually stored. The main idea is



that we can compute an index from each key to be inserted, in $O(1)$ and use that index as the index of the array location where to store the object. Obviously a because the number of slots is less than the number of possible keys it is possible for two key to be stored at the same location hyelding to a *collision*. There are various effective approach to handle collision which will be describer in the following sections. In a hash table an element with is stored at slot $h(x)$ where h is an hash function which maps the universe of keys U into slots of the hash table.

$$h(x) : U \mapsto \{0, 1, \dots, m-1\}$$

with m typically much smaller than $|U|$. The hitch is that (pigeone priciple) two object may be assigned the same slot (they hash to the same slot). This situation is called a **collision**. There are several tecnquiques to deal with collisions such as

1. chaining
2. open addressing
3. linear probing
4. quadratic probing
5. random probing

Of course the best scenario is to avoid collision altogether but since the size of the universe if bigger than the size of the hash table there always exists at least a couple of object which will hash to the same slot. The hash function should, in order to minimize collision as much random as possible (remaining deterministic obviously). The word hash means to mix and it somehow evoke what this function does. It mixes data taken from the object in order to provide a random looking index to be used as an index of the hash table.

11.0.2 Collision resolution by chaining

In chaining, all the elements that hash to the same slot are placed into the same linked-list. The hash table can be viewed, infact, as an array of linked list where each node of the list contains apointer to the object stored. Dictionary operation can so be performed in average case $O(1)$ as follows:

ANALYSING PERFORMANCE OF HASHING WITH CHAINING

Given an hash table T we define the **load factor** ad the ration $\alpha = \frac{n}{m}$ where n is the actual number of element stored into the hash table and m is the dsize of the table. The worst case scenario for

```
Procedure SEARCH (T,x)
    $LIST-SEARCH(T[h(x)]$ /* Worst case - O(n) we will see that average
        case is much better O(1)                                          */
Procedure INSERT (T,x)
    $LIST-INSERT-HEAD(T[h(x)],x)$ /* O(1)                                 */
Procedure DELETE (T,x)
    $LIST-DELETE(T[h(x)],x)$ /* O(1) is list are doyhbly linked or O(n)
        if singly linked                                                  */
```

Algorithm 8: Hash table - chaining dictionary operations

hashtable with chaining is very bad. When all the inserted element hash to the same slot the seach procedure degrade to linear search in a list, with complexity $\Theta(n)$. We will show now that hash table performs much better on the average case and when certain operation such as enlarging or shrinking are performed carefully. The main question here is: what is the complesxity for a hash table lookup operation? We evaulate the complexity of this operation with the length of the list being searched. A lookup operation may be succesful or unsuccesful. Analysing the former case is easy, since assuming that object are hashed with a **simple uniform hash function**[1] while the latter requires a bit more work. Infact if we denote the length of the list $T[k]$ at location k as n_k then $n = \sum_{i=1}^{m-1} n_i$. The expected value for n_j is then α.

In a unsuccesful lookup operation the list $T[j]$ must be traversed completly from head to tail giving so a complexity of $\Theta(1+\alpha)$ (computing of j via the hasing function, plus the length of the list being searched). In a succesful lookup operation the list being search will be searched half way through on average case so the complexity will be $\approx \Theta(1+\frac{\alpha}{2})$. Another approach would be to consider that supposing that the element being search lie in at location j of list $T[l]$ all the element at indices $j-1, j-2, \dots, 0$ are inserted **after** the element being searched (because the insert operation push the element at the head of the list. In a succesful search then we examine all the element that have been inserted before. What is the probability that two element hash to the same slot? Again, $\frac{1}{m}$ because of the simple uniform hashing assumpion. So the cost of a single lookup is 1 plus the number of element inserted after j that hash to the same location of element j:

$$1 + \sum_{j=i+1}^{n} = \frac{1}{m} = 1 + \frac{1}{m}\sum_{j=i+1}^{n} 1 = 1 + \frac{1}{m}(n-j)$$

Taking the average over n insertion we have

$$\frac{1}{n}\sum_{i=1}^{n}(1+\frac{1}{m}(n-j)) = \frac{1}{n}(n+\frac{1}{m}\sum_{i=1}^{n} n - \sum_{i=1}^{n} i) = 1 + \frac{1}{nm}(n^2 - \frac{n(n+1)}{2}) = 1 + \frac{n^2}{nm} - (\frac{n^2}{2} + \frac{n}{nm}) = 1 + \alpha - \frac{\alpha}{2} + \frac{1}{2m}$$

which is $\Theta(1+\alpha)$

What does this mean? That if we can keep the value of α **constant** then we can perform all the dictionary operation in $O(1)$ average case! How can we keep α constant? If we keep m is of the same order of n then we have $\alpha = O(m)/n = (1)$ This tells us that the size of the hash table should be of the same order of the number of element stored in the has table itself. If we can ensure this invariant then dictionary operation can be performed in **constant time**!

Alternative to Linked Lists

Chaning with linked list in not obviouly the only possible solution. If we know that out universe is totally orderable than we could store the collisions in a binary search tree insted lowering the complexity from $\Theta(1+\alpha)$ to $\Theta(a + log(\alpha)$.

[1] A function that hashes each object to one of the slots with uniform probability $Pr(h(obj) = j) = \frac{1}{m}$ and independently of the already hashed objects

11.0.3 Hash Functions

A good hashing function satisfies the assumpion of simple uniform hashing. This condition is not easy to ensure a priori and is sometimes impossible to check because we rarly know the probability distribution of the key drwan. More importantly this keys may not be drawn independenlty. Heuristic are emplaced in the design of good hashing functions. For example in a compiler symbols table, it would be good for two similar words to hash in different slots (since for example two symbols like `pt` and `pts` may often occur in the same listing). Keys and object are often interpreted as natural numbers (and an important part of the hashing process is how we interpret the object as natural numbers. For instance a string may be interpreted as the sum of the ASCII values of all the character[2]). This key is then passed to another function which maps the keys into the most appropriate slot in the hash table.

Divison Method

In the division method a key k is mapped into one of the m slots by taking the reminder of k divided by m (usually $k > m$)

$$h(m,k) = k \, mod \, m$$

This methos is quite fast since it only requires a division. When using the divison method the size of the table, m should not be a power of two since the division of $\frac{m=2^p}{k}$ consist of the p-lowest bits of k. For example let $m = 2^9 = 256_{10} = 010000000_2$ and $k = 300_{10} = 100101010_2 \mapsto k \, mod \, m = 000101010_2$ With this method the best choice is prime number. But choosing m prime does not solve all the problem or makes this method *perfect* algtogether. All integer of the form $x + am$ will hash to the same slot x! Is quite easy then for a malicious agent to come up with a bad set of values that all hash to the same bucket.

Multiplication Method

The multiplication method operates in two steps:

1. We multiply k by a constant (called **salt**, chosen at creation time and remaining fixed for the entire lifetime of the hash table) $0 < A < 1$ and extract the fractional part of the result.
2. We then multiply this value by m and take the floor of the result.

$$h(m,k) = \lfloor m(Ak \, mod \, 1) \rfloor$$

For instance suppose the key is 102 and $A = \phi \approx 0.6180339887$ and $m = 30$. The bucket is computed as follows:

$$30(\phi * 102 - \lfloor \phi * 102 \rfloor) = \lfloor 30(63.0394668474 - 63) \rfloor = \lfloor 30 * 0.394668474 \rfloor = \lfloor 11.84005422 \rfloor = 11$$

Since the fractional part of Ak is a number always smaller than one, $h(m,k)$ is unsured to be in the interval $[0, m-1]$. A good advantage of this method is that the value of m is not critical and infact we often choose m to be a power of two since we can easily implement the method using shifts operators.

1. choose $m = 2^p$
2. multiply w bits of k with w bits of $(A * 2^w)$ and we obtain a 2^w word number.
3. Extract the first p bits of the lower hald of this product.

[2]Note that this is a terrible interpretation for a string since all the permutation of the same string willl produce the same key value!

A good choice as suggested by Knut is the golden ration ϕ.

Another good example in this class of hash function was analyzed by *Dietzfelbinger et al.* in 1997, called binary multplicative hashing. The goal is to hash w-bit integers (word size integer) to l-bits (label) integers. we define such function as

$$d_a(x) = \left\lfloor \frac{(ax) mod 2^w}{2^{w-l}} \right\rfloor$$

with $a \in \{1, 2, \dots m-1\}$

```cpp
typedef unsigned int uint;
#define INT_SIZE (32)
 int hash(const uint m,const  uint k, const uint a, const uint hash_size) {
  return (a*k) >> (INT_SIZE-hash_size)
  //division by powers of 2 is equivalent to a right shift
 }
```

Listing 11.1: "Multiplicative method hashing"

Note that the product ak may be a $2^{2 \times INT\_SIZE}$ number, but this would cause overflow, so only the lower `INT_SIZE` of the resulting product will be kept. Right shifting by the right amount will do the job of isolate the inner l portion of the number. For instance consider the following listing:

```cpp
#include <iostream>
using namespace std;
typedef unsigned int uint;
int main() {
  uint a = 523456;
  uint res= a*a; //overflow
  cout<<(uint)res<<" "<<a<<endl;
  return 0;
}
```

`res` variable contains the number 3423244288 which is the wrong answer since $523456^2 = 274006183936$. 3423244288 correspond is a 32-bit integer which corresponds to the first 32-bits of the 64-bits number 274006183936. $3423244288 = 274006183936 mod 2^{32}$.

Universal Hashing

Any possible scheme is vulnerable to the fact that is alway possible to choose n objects whose keys maps to the same bucket, yelding as we know to the very bad $\Theta(n)$ lookup. The only approach that overcome this issue is for the hashing function to hash keys *randomly* and *indeentently*. In unversal hashing we select at the beginning of the the execution (at hash table creation time if working in a object oriented programming environment) a suitable hash function from a set well designed class of functions. We will use randomizaion in the costruction of the hash function in ordder to ensure that for any sequence of element inserted in the table the expected performance will be good (i.e. the horrific $\Theta(n)$ lookup would occuper with a very little probability). This is the same kind of expectation quicksort assumes when choosing the pivot element.

To summarise when the keys to be inserted are uniformly drawn from a universe of keys is not difficould to come up with an hash function that distribute them uniformly across all the buckets meaning that the probability of collisions between any of two keys is the $\frac{1}{m}$ (any bucket at any time is equally likely to be chosen by the hash function). The main caveau is that the any assumption on the distribution of the keys to be inserted is not realistic manily because in real life keys are not drawn randomly, then because there could alway be the case then an adversary would attack our system submitting a patological sequence of keys which will hash to the same bucket. So a much *more approach is to make no asumption on the distribution of the keys and let randomess come into play in the definition of the hash function itself which is not now fixed. Looking now at the average running time of all the hash table operations withouth, again any assumpion on the distribution of the keys. An adversary cannot find a* ***bad sequence of keys*** *since the hash function is not known* ***a priori****. Randomess, which willl ensure good average case behaviour is in the definition of the hash function!*

A family of function $\mathbb{H}$ is a **universal hash function family** if each function the probability that two element collide is at most $\frac{1}{m}$ i.e. given two elements $x, y \in U$, $\forall h \in \mathbb{H}$ we have $Pr(h(x) = h(y)) \geq \frac{1}{m}$ (Sometimes this probability may be $O(\frac{1}{m})$ and the family is called ε-almost universal). Not that universality does not implies uniformity[3]. Note also that uniformity is not terribly useful and is not a string enough condition to ensure good hash table performance on average. Take for instance the set $\mathbb{K} = \{k_a(x) = a \, : 0 \geq a < m\}$ of constant function. This set is perfectly uniform in the sense that picking a random function k_a at random each slot is equally likely to be choosen as destination slot. Probability that object x hashes to slot l is the probability that function k_l is picked up at random from the set $\mathbb{K}$ which clearly is $\frac{1}{m}$. That is why minimizing the number of collision is much more important. The set $\mathbb{K}$ is uniform but is not unversal (according th the following definition) because the probability that picking up a random function k_a and given two keys x, y they hash to the same location is not less than $\frac{1}{m}$ but is 1.

Lineary of expectation

By linearity of expectation we mean that the **expected value** of the sum of a random variable is equal to the sum of their individual expected values. It is essentially a weighted average of possible outcomes. Suppose we are interested in the expected value of the sum of K dices. Calculating the the sum the usual way is tedious. We have to sum up all possible outcomes and divides by the total number of outcomes to get the expected values.

				SUM
(1,1)	*(1,1)*	...	*(1,6)*	$2+3+4+5+6+7 = 27$
(2,1)	*(2,1)*	...	*(2,6)*	$3+4+5+6+7+8 = 33$
(3,1)	*(3,1)*	...	*(3,6)*	$4+5+6+7+8+9 = 39$
(4,1)	*(4,1)*	...	*(4,6)*	$5+6+7+8+9+10 = 45$
(5,1)	*(5,1)*	...	*(5,6)*	$6+7+8+9+10+11 = 51$
(6,1)	*(6,1)*	...	*(6,6)*	$7+8+9+10+11+12 = 57$

expected value is $\frac{57+51+45+39+33+27}{36} = \frac{252}{36} = 7$. Linearity of expectation says that the expected value of that sum of two dices is the sum of the expected value of the single dices which is $\frac{1+2+3+4+5+6}{6} = 3.5$. The expected value of the two dices is then 3.5×2. This also works if the dices have different expected values. Imagine a dice with number 3 substituted by number 99. The expected value of the sum of the two dices is then $3.5 + 19.5 = 23$.

Surprisingly this also works when the variables to be summed are **not independent**.

Suppose we want to find the expected value of the sum of two random variables X and Y.

$$E[X+Y] = \sum_x \sum_y (x+y) \times Pr(X = x, Y = y) =$$

then using associativity,

$$\sum_x \sum_y x \times Pr(X = x, Y = y) + \sum_x \sum_y y \times Pr(X = x, Y = y) \implies$$

[3]The propoerty such that all the possible hash values are equally likely to be returned. $Pr(h(x) = z) = \frac{1}{m}$

$$\sum_x x \sum_y Pr(X=x,Y=y) + \sum_y y \sum_x Pr(X=x,Y=y) \implies$$

$$\sum_x x \sum_y Pr(X=x) + \sum_y y \sum_x Pr(Y=y) \iff E[x]+E[Y]$$

since $Pr(X=x,Y=y) = Pr(X=x)$ in the first part of the equation because summing up all the probabilities where X=x and Y can be whatever number gives the probability of $X=x$. Same logic applies to $Pr(X=x,Y=y) = Pr(Y=y)$ in the second part of the formula[4]. Using induction on the number of variables this resuolt can be extende to an arbitrary number of variables. Continuous variables can be handles substituting summations with integrals.

■ **Example 11.1 There is a group on n men all with a single hat on. The hats are redistribuited atrandom to every man. What is the expected number of hats each man get?** This problems is easily solvable by means of linearity of expectation. Let N_i be the number of hats received by each man and C_{ij} and indicator variables which is 1 if and only of hat i goes to man j.

$$E[N_j] = E[\sum_i^n Cij] = \sum_i^n E[Cij] = \sum_i^n Pr(C_{ij}=1)$$

Since hats are uniformly distributed the probability that hat i goes to man j is $\frac{1}{n} \Rightarrow \sum_i^n \frac{1}{n} = 1$. ■

■ **Example 11.2 If the sum of two numbers rolled on the dice is A and their product id B, compute E[A+B]**

We know that the expected value of the sum of two numbers rolled is $A = 7$. In order to compute B we apply linearity of expectation. What is the expected value of a single dice? 3.5, so by linearity of expectation 3.5 $times$3.5 $= 12.25$. The soltuion is then: $E[A+B] = E[A]+E[B] = 7+12.25$. ■

■ **Example 11.3 The digits $1,2,3,4$ are randomly arranged to form two two-digit numbers AB, and CD. What is the expected value of the product $AB \times CD$?**

We have to try to write the product as some kind of sum in order to apply the principle of linearity of expectation. We should note that $AB \times CD = (10A+B) \times (10C+D) = 100AC + 10AD + 10BC + BD$ This is an easier problem because by lineary of expecation we can sum up all the expected values for the product of the single digits AC, AD, BC, BD. What is then

$$E[AC] = E[AB] = E[BC] = E[BD] = \frac{1\times2+1\times3+1\times4+2\times3+2\times4+4\times3}{6} = \frac{35}{6}$$

The final value is now easy to compute:

$$100E[AC] + 10(E[AD]+E[BC]) + E[BD]) = 121 \times E[AC] = 705,8\overline{3}$$

This approach is much easier than computing the expected value listing explicitely4! possible couple of two-digits numbers. ■

■ **Example 11.4 Let's take another example. Suppose you have a bag with 4 balls, each of different color. You have to select 4 balls from the bag (with replacement). What is the expected number of different colors that will be extracted?** in order to use the principle of linearity of expentancy, we should write the expected number of different colors as a simple sum of random variable for which is easy to compute their expected values. Suppose now for a moment the following set of balls has been extracted: **Red, Blue, Blue, Yellow**. If we set a indicator variable P_c =<*color c has been extracted*> then $C_{green} + C_{blue} + C_{red} + C_{yellow} = 0+1+1+1 = 3$. We have reduced the problem as a sum of simple to compute Indicator variable which expected value corresponds to the probability that the condition for which the indicator takes value one is true. What is the probability that a color is selected? Is 1 minus the probability is not selected. More specifically, $Pr(C_c = 1) = 1 - (\frac{3}{4})^4 = \frac{175}{256}$

Solution to the problem is then $E[C_{green}] + E[C_{blue}] + E[C_{red}] + E[C_{yellow}] = 4 \times \frac{175}{256} = 2,734375$

What if we have a bag with n balls of different colors and we have to extract n of them? How many different colors we are expected to extract? Same principle : $Pr(C_c = 1) = 1 - (\frac{n-1}{n})^4$ Solution would be $n \times (1 - \frac{n-1}{n})^n$ ■

■ **Example 11.5 25 independent, fair coins are tossed in a row. What is the expected number of consecutive HH pairs? If 6 coin tosses in a row give HHTHHH, the number of consecutive HH pairs is 3.** There are 24 consecutive pairs: $HHH\dots H$ Let use the indicator variable $C_i = 1$ if pair i is HH. Probability that C_i is 1 is $\frac{1}{4}$. $E[\sum_{i=1}^{2} 4C_i] = \sum_{i=1}^{2} 4E[C_i]$ by linearity of expectation.

$$\sum_{i=1}^{24} \frac{1}{4} = \frac{24}{4} = 6$$

So the expected number of consecutive HH is 6.

Note that we could have solved the same problems noticing that: if we denote p(n) the number of pairs in n extraction then, p(1) = 0; Successive extraction works as follows:

- Extract H where previous coin was H (probability$\frac{1}{4}$). This add one pair
- Extract H where previous coin was T Or extract T with whatever previous coin leave the number of consecutive head pair unchanged (probability$\frac{3}{4}$, three cases HT,TH,TT).

We can write this as recurrence formula:

$$p(1) = 0; P(n) = \frac{3}{4}p(n) + \frac{1}{4}(p(n)+1) = p(n) + \frac{1}{4};$$

Expanding $p(n)$ up to $n = 25$ gives : $\frac{1}{4} + \frac{1}{4}\ ldots + \frac{1}{4}$(24 times) ■

■ **Example 11.6 Coupon Collector: Suppose there are *n* types of coupons in a lottery and each of them is equally likely to get extracted. What is the expected number of lots that have to be bought until we have at least a copy of *all* coupons?** Before to dive into the main problem is useful to stop and think to an easier version of this problem. Suppose the number of coupons is two. Let's think to the lottery as having three state:

1. No coupons have been extracted.
2. One coupon has been extracted (This state is always reched after 1 extraction).
3. Both coupons have been extracted.

State number 1 only lasts for a trial while number 2 is kept until we keep extraction the first coupon. Let's denote by X_i the number of extraction in state i. What we are really interested is $E[X_0]$. Obviously $E[X_2] = 0$ since once we reach that state we don't need to extract any other coupon. What is the probability of a coupon to be extracted? $p = \frac{1}{2}$.

$E[X_0] = 1 + E[X_1]$. $E[X_1] = 1 + (1-p)E[X_1] + pE[X_2] = 1 + E[X_1] - pE[X_1] + 0$

$$E[X_1] = 1 + E[X_1] - pE[X_1] + 0 \Rightarrow 1 = pE[X_1]$$

Hence $E[X_1] = \frac{1}{p} = 2$ This gives us an important insight. Whenever we switch from state i to j and and the switching probability is p the expected number of trials is $\frac{1}{p}$. Going back to the original problem. The number of state is n cornensponding to having $1,2,3\dots,n$ different coupons. The idea is that if we know what is the expected number of trials for going from states $1 \to 2$ and then from states $2 \to 3$ and so on up to the last transition $n-1 \to n$ then $E[X_0] = 1 + E[X_1] + E[X_2] + E[X_3] \dots + E[X_{n-1}]$ What is then the probability of going from state i to j? If we are in state i we have collected i coupons and we extract a *new* one with probability $1 - \frac{i}{n}$ (Certainity minus the probability of extraction one of the already extracted coupons). We have already seen that given this probability $E[X_i] = \frac{1}{1-\frac{i}{n}} = \frac{n}{n-i}$

$$E[X_0] = 1 + \sum_{i=1}^{n-1} \frac{n}{n-i} = 1 + \Big(\frac{n}{n-1} + \frac{n}{n-2} + \frac{n}{n-3} + \dots \frac{n}{n-(n-1)}\Big)$$

$$= 1 + n\Big(\frac{1}{n-1} + \frac{1}{n-2} + \frac{1}{n-3} + \dots 1\Big) = 1 + nH_n$$

Let's for instance fix $n = 12$. The expected number of trials is then $1 + 12 \times (\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{11}) = 1 + 36.2358\overline{281385} = 37.2358\overline{281385}$ ■

Given a subset of $S \subseteq U$ of size n, $\forall x \in U$ the **expected** number of collisions between x and other element in S is at most $\alpha = \frac{n}{m}$. The total number of collision between x and any other element from S is $\sum_{y \neq x} Pr_{xy} < \frac{n}{m}$ by lineary of expectation. The expected value of the number of collision between x and other elements of S is the sum of the expected value for the collision between x and the other elements. This means that the expected number of elements in each slot is α!

The proof to the above statement can be outlined as follows: Let C_x be a random variable indicating the total number of collision of keys in S with x and C_{xy} be a random indicator variable s.t. $C_{xy} = 1 \iff h(x) = h(y)$ The espected number of collision of x is the expected sum of all the indicator variable C_{xy}.

$$E[C_x] = E[\sum_{y \in S \setminus x} C_{xy}]$$

By lineary of expectation, the expected value of a sum of a random variable is the sum of the expected values of each random variable so,

$$E[C_x] = \sum_{y \in S \setminus x} E[C_{xy}]$$

The expected value of an indicator random variable is its probability to be one. By definition of universal hashing we already know that C_{xy} is one whenever a collision occurs, i.e. with probability $\frac{1}{m}$.

$$E[C_x] = \sum_{y \in S \setminus x} \frac{1}{m}$$

There are $n-1$ element in $S \setminus x$,

$$E[C_x] = \frac{n-1}{m} < \frac{n}{m} = \alpha$$

11.1 Open Addressing

In open addressing each element is stored directly into the table itself. This mean that the load factor can't be greater than one cause at some point the entire table may be filled completly. Each slot of table contains either a valid entry or a a special value `NIL` which indicases whether the slot is empty or not. It may also contains other flags, one of which is worth to mention now, `DELETED` which identifies slots which have been cleared and may lay in the middle of two valid stored objects. One of the main advantages of open addressing versus chaining is that the memory footprint is less important cause there are no pointers at all to be managed since the object are stored into the table itself.

The key idea behind open addressing is that a the hash function is extended to take an additional parameters, the probe number. Changing the probe number cause the hash function to return a different slot index with the property that eventually all the slots are returned. This is an essential property that such function need to have in order to be suitable for open addressing use. The hash function need to return a permutation of all the slot indices as the probe number changes.

$$\{h(k,1,s),h(k,2,s),\dots,h(k,m,s)\} \Rightarrow \{a_i \,|a_i \in \{1,2,\dots m\}\}$$

where h is the hash function the second parameter is the probe number and s is the size of the table.

$$h : U \times \{1,2,\dots,m\} \to \{1,2,\dots,m\}$$

This condition is required because of the way dectionary operation are implemente in open addressing.

Insertion $I(k)$ is performed as follows: starting with probe $p = 1$ we iteratively try to insert the element with key k into the slot $h(k, p = 1, m) = v_p$. If it is empty i.e. it contains NIL or DELETED flag we can proceed to store the element in slot v_p. If it is already used to hold another object we keep increasing the probe number (which produces a new slot index) and try the insertion again. Eventually all the slots are probed cause the way the hash function is designed (produces a permutation of all the indices). If $h(k,m,m)$ fails than this means that the table is **full** and a *enlarge* operation need to be performed.

```
Procedure INSERT (T,x)
    $p \leftarrow 1$;
    while $h(k,p,m) \neq NIL \wedge h(k,p,m) \neq DELETED$ do
        $p \leftarrow p+1$;
    end
    if $p == m$ then
        return TABLE OVERFLOW ERROR;
    end
    $T[h(k,p,m)] = k$;
```
Algorithm 9: OPEN ADDRESSING INSERT

Search works as follows: since probes number are unrolled sequentially during insertion it is clear that whenever we found an empty slot on our searching process this means that the element is not present in the table at all, cause if it was it would have been inserted into the first empty slot probed by the same function we are quering during search.

Deletion is very similar to search, infact it uses it as subroutine. key k is first searched, and if it does not exists into the table then nothing else happens (you can't delete something that does not exist). If a valid slot is returned by the seach routine then the slot cannot be marked as NIL cause this would cause search to fail. Imagine the element k_1 has been inserted at location $l_1 = h(k_1,3,m)$

```
Procedure SEARCH(T,x)
    $p \leftarrow 1$;
    while $T[h(k,p,m)] \neq k \wedge T[h(k,p,m)] \neq DELETED \wedge p \leq m$ do
        $p \leftarrow p+1$;
    end
    if $T[h(k,p,m)] == k$ then
        return $h(k,p,m)$;
    end
    return NIL;
```
Algorithm 10: OPEN ADDRESSING SEARCH

at time t_1. Imagine that another element at time $t_2 > t_1$ is inserted. It happens that is inserted at location $h(k_2,2,m) = l_2 \neq l_1$ at the second probe cause it happens that at the first probe the slot was busy, $h(k_2,1,m) = h(k_1,3,m)$. When element k_1 is deleted and the corresponding slot marked as NIL then element k_2 becomes inaccessible since search operation would return NIL when it first probes to search for it at location $h(k_2,1,m)$ which has been nulled by delete routine when probing $h(k_1,3,m)$. Here the reason why we need another flag to be used when deleting an element. During search DELETED slots are treated as not empty and the search is not stopped. Insert operation treat them as empty slots so they are reused to hold new values. Note that because of how deletion procedure works, search times may not depend on the load factor α anymore so when deletion is required **chaining** is usually prefereable since it gives more predictable performance. Infact even at low load factors is still possible to have bad search performance due to **clustering**.

So the question that raise naturally is the following: **how can we generate probing sequences that perform well?** There are various strategies such as **linear,quadratic or double** hashing.

Linear Probing

Let's start saying that this strategy is all way bad and should never be used. It is showed here only for learning purposes. It creates long clustered sequence that raise the searching time even at very low load factors.

Linear probing uses an additional function $h' : U \rightarrow \{1,2,\dots,m\}$ that select the starting point of the linear sequence defined as follows:

$$h(k,p,m) = [h'(k) + p] \bmod m$$

The generated sequence is a linear sequence wrapped around the value m and starts at $h'(k)$. Since there are only m starting points (and it is the only thing that differentiate a sequence from another) the possible sequences are m only. Despite it is very easy to implement it suffers from what is called **primary clustering**. Clusters arise because the probability of a slot preceeded by i filled slots is proportional to i itself! This means that long sequences of full slots are more likely to become even longer. Assuming uniform hashing, infact, a slot l can be filled during insertion for two reasons:

1. the probing sequence starts at l i.e. $h(k,1,m) = l$ with probability $\frac{1}{m}$ since key k has uniform probability of being mapped to any of the slots (uniform hashing hyphotesis).
2. the insertion procedure starts in one of the preceeding and already filled slots. It will return the first empty slots it encounters in the probing sequence which is l. So all the elements which maps into one of the preceeding filled slots will end up in being inserted into slot l. How many of them? i. Since are all independent events proibability sum up.

What does that mean? It means that a slot has probability $\frac{1}{m} + \frac{i}{m} = \frac{i+1}{m}$ to be choosen as inserting index.

R **Linear probing is bad. It suffers from primarry clustering problem. Long clusters are more likely to become even longer.**

Quadratic Probing

This probing strategies uses a quadratic formula to compute the probing sequence. It is supported by a function h' defined as in linear probing and two more parameters c_1, c_2. The quadratic function hash function family is defined as follows:

$$h(k,p,m) = [c_1 p^2 + c_2 p + h'(x)] \mod m$$

Thi approach works much better in practice than linear probing but it also suffer from clustering. Its form of clustring is called secondary clustering. Since the initial probaing value $h'(x)$ defines the entire sequence only m probing sequences are possible[5]. This strategy is aslighty more challenging to be implemented since parameters have to be selected s.t. the hash function make full use of all table slots. How c_1, c_2 have to be choosen s.t. h will return the permutation of $\{1, 2, \dots, m\}$ as the probing number changes?

Exercise 11.2 Suppose we have to search for a key k in a hash table of size m (indices start from 0) and we are provided with an hash function$h : U \times \{0, 1, \dots, m-1\}$. The search scheme works as follows: Shows that

```
Procedure SEARCH(T,k)
    $i \leftarrow 1$;
    $j \leftarrow h(k)$;
    do
        if $T[j] == k$ then
            return $j$;
        end
        if $T[j] == NIL$ then
            return $NIL$;
        end
        $i \leftarrow i + 1$;
        $j \leftarrow (j + i) \mod m$;
    while $i < m$;
```

- This scheme is infact an instance of the more general *quadratic probing* and exibith the appropriate parameters c_1, c_2
- It is a legal hash function s.t. it examines all the slots eventually.

■ **Solution 11.2** Let's first notice that given a starting point for the sequence $j_0 = h(x)$ the subsequent slots indices are computed as using the following recurrence function

$$j_{i+1} = (j_i + i) \mod m$$

The k^{th} index is then $j_k = (((j_0 + 1 \mod m) + 2 \mod m) + 3 \mod m) \dots + k) \mod m$ which is equivalent to

$$(j_0 + 1 + 2 + 3 + \dots + k) \mod m$$

In order to prove that let's show that

$$((j_0 + i) \mod m) + i + 1 \mod m) = (j_0 + i + i + 1) \mod m$$

[5] $h(k_1, 0, m) = h(k_2, 0, m) \implies h(k_1, i, m) = h(k_2, i, m) \iff h'(k_1) = h'(k_2)$ since $c_1 i^2 + c_2 i$ is always the same.

Now, $((j_0+i) \mod m)+i+1 \mod m) \neq (j_0+i+i+1) \mod m \iff (j_0+i) \mod m) \neq j_0+i$ This can only be true if and only if $j_0+i \geq m$. This implies $\Rightarrow i \geq m-j_0$ So assuming $i \geq m-j_0$ then $((j_0+i) \mod m) = i-m+j_0$. The original equation becomes then:

$$(i-m+j_0+i+1 \mod m) = (j_0+i+i+1) \mod m$$

which is true since adding or subtracting m in any operation modulo m does not change the result at all. Now let's rewrite $(j_0+1+2+3+\ldots+k)$ mod m as $j_0+\frac{k(k+1)}{2}$ mod $m \Rightarrow j_0+\frac{1}{2}k\frac{1}{2}k^2$ mod m This is an instance of the quadratic probing formula with $c_1 = c_2 = \frac{1}{2}$.

Let'a now prove that all the slot indices are actually probed. In order to do that let's first notice that each probing sequence is at maximum of length m and m values should be probed. This means that no two different probes can output the same number. This is equivalent of proving the following:

$$(j_0+1) \text{ mod } m \neq j_0+1+2 \text{ mod } m \neq j_0+1+2+3 \text{ mod } m \ldots \neq j_0+1+2+\ldots+m-1 \text{ mod } m$$

$$j_0+\frac{l(l+1)}{2} \mod m = j_0+\frac{k(k+1)}{2} \mod m \Rightarrow \frac{l(l+1)}{2}-\frac{k(k+1)}{2} \mod m = 0 \Rightarrow \frac{l^2}{2}+\frac{l}{2}-\frac{k^2}{2}-\frac{k}{2} \mod m = 0$$

11.1.1 Double Hashing

Doble hasing is one of the best methods available for **open addressing** since it produces much more random-like probing sequences. it uses an hash function of the following type:

$$h(k,i) = (h_1(k)+ih_2(x))modm$$

where h_1 and h_2 are auxilary functions. The reason why double hashing performs better than linear and quadratic probing is that the final hash values depends twice from k, since both the first number of the sequence and the offset depends on k. It is important to note that the value of $h_2(x)$ need to be relatively prime with m and it usually force to be like so by choosing $m = 2^l$ and $h(x)$ to return always an odd number. The number of possible functions is $\Theta(m^2)$ since each pair of (h_1, h_2) yeald to a different probing sequence. Although in principle other number than m prime or m baing a power of two can be used, in practice it is quite hard to wensure the co-primality of m and h_2 since the ratio $\frac{\phi(m)}{m}$ can be small[6].

11.1.2 Analysis of Open addressing Performance

[6] ϕ is called totient function and counts the number of number which are relatively prime to m and h_2 can only map x into one of these numbers

Bibliography