

Algorithms and Data Structures

Saul Pierotti

April 1, 2020

Introduction

- Exam will be written, but you can do oral if you want
 - After lectures, around June 25
- An algorithm is a finite series of steps that solves a problem
 - In CS, an algorithm is a well defined computational procedure
- For a sorting algorithm, the input is a series of numbers and the output is an ordered series of numbers
- Algorithms are for humans, while a program is for a computer
- Algorithms are written in pseudocode, which follow specific conventions
- A problem can be solved by many algorithms
- An algorithm can be implemented in many different programs
- Properties of algorithms
 - Input for an algorithm can have 0 or more inputs
 - It always has 1 or more outputs
 - It should be clearly defined and unambiguous
 - It should terminate after a finite number of steps
 - All operations must be basic
 - * They can be solved exactly and in finite time
- The correctness of an algorithm is difficult to prove
 - I would need to try all possible inputs (!)
 - Published algorithms have a mathematical proof
- Incorrect algorithms can produce a wrong output or not produce any for some instances
 - In some cases they are still useful, if I can control their error rate
- Efficiency is related to the ability of an algorithm to be executed with available resources
- Resources are time and memory
- Time is measured in running time, not CPU time
 - CPU time is dependent on CPU (!)
 - CPU time is number of instructions divided by number of instructions per unit time
 - Running time is the number of primitive operations to be performed in proportion to the input size
- The algorithm influences time much more than hardware, we do not focus on hardware (!)
- Running time of n^2 is unacceptable for large inputs
- Using the right data structure is important for efficiency
- Decision trees are essential in CS
- An instance of a problem is a specific input for that problem
- In a while and for loop, the test is always executed once more than the body

Math background

- Finite sums
 - $\sum_{k=1}^n a_k = a_1 + a_2 + \dots + a_n$
- Infinite sums

- $\sum_{k=1}^{\infty} a_k = a_1 + a_2 + \dots$
- Sums are linear
 - $\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$
- The arithmetic series
 - $\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$
- The quadratic arithmetic series
 - $\sum_{k=1}^n k^2 = 1 + 4 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$
- The cubic arithmetic series
 - $\sum_{k=1}^n k^3 = 1 + 8 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$
- The geometric series
 - $\sum_{k=1}^n x^k = 1 + x + x^2 \dots + x^n = \frac{x^{n+1}-1}{x-1}$
 - When x is less than 1
 - * $\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}$
 - * $\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Other formulas
 - $\sum_{k=1}^n \log k \approx n \log n$
 - $\sum_{k=1}^n k^p \approx \frac{n^{p+1}}{p+1}$
- A set is a non-ordered and non repetitive collection of elements
- Sets can be infinite
- It is described as $S = \dots$
- 2 sets are equal if they contain the same elements
- The cardinality of a set $|S|$ is the number of elements it contains
- If A contains all the elements contained in B and also other elements, then B is a proper subset of A
- The power set $P(S)$ is the set of all subset of S , including the empty set and S itself
 - $|P(S)| = 2^{|S|}$
- The cartesian product of 2 sets is a set containing all possible pairs of elements
- The cartesian product of n sets is a set of n -tuples
- A tree has only one way to go from one node to the other
- A graph can have cycles
- A forest is made of many trees
- Any non-empty tree with n nodes has $n-1$ edges
 - If this is not true, we don't have a tree
- A tree is rooted if one of its nodes is distinguished as root
 - It can be defined recursively such that every non-root node of a rooted tree is itself the root of a subtree
- Tree terminology is similar to that of ancestry trees
- The depth of a node is its distance from the root (number of edges)
- The height of a node is the lenght of the longest path to a leaf
- A binary tree is an order tree with 2 subtrees wich are themselves binary

Pseudocode

- We start counters from 1 since it is easier to understand
- Bold words are reserved words like **return**
- Variables are always local to the current procedure
- We can have loops like **while** and **for**

for $i=0$ to/downto $i=4$ (by 3)

 <statement>

- We have if statements

if <condition>

 <statement>

- Comments are rendered with `//`
- No colon/semicolon at the end of lines
- Use of indented blocks
- Differentiate conditional expressions and assignments (!)
- A slice of an array is indicated as `A[3..5]`
- Attributes of objects are indicated as `object.attribute`
 - The length of an array can be indicated as `A.length`

Sorting

- Sorting is an intermediate step in many tasks in CS
- There are many sorting algorithms

Insertion sort

- It is like arranging card in order in your hand by picking one at a time
- I take 1 unsorted object at a time and I insert it in the correct position in the sorted array
 - I compare with all the objects in the sorted array, until I find the right position
- I start from the first element of the array and I don't do anything
- I take the second element, and if it is smaller than the first I swap them
- I take the third, and if it is smaller than the second I compare it with the first and I swap in the right position
- I continue like this for all the elements

Pseudocode

```

INSERTION-SORT(A)
  for j = 2 to A.length
    key = A[j]
    i = j-1
    while i > 0 and A[i] > key
      A[i+1] = A[i]
    A[i+1] = key

```

Running time

- Nearly sorted numbers can be sorted much faster with insertion sort
- The input size is the length of the array
 - $n = A.length$
- The initial **for** test is executed n times
 - It is n , not $n-1$ because even when it is false we still have to check ones (!)
 - The body of the **for** is executed $n-1$ times
- The assignment of `key` is therefore executed $n-1$ times
- The **while** test is executed $\sum_{j=2}^n t_j$ times
 - The body of the **while** is executed $\sum_{j=2}^n t_j - 1$
 - There are 2 assignments on the while body
- The final assignment after the **while** inside the **for** is executed $n-1$ times

Best case

- The array is already sorted
- I never enter the while, but I do completely the for
- This means that t_j is 1, I only do the test
- The time is linear

Worst case

- The array is in reverse sorted order
- The time is quadratic

Average case

- It is really difficult to do, we prefer to focus on the worst case

Evaluation

- For almost sorted sequences its running time is almost linear
- Can be online
 - It can sort sequences as they arrive
- In the worst and average cases it is quadratic
 - Quadratic is really bad (!)

Merge sort

- It is a divide and conquer algorithm
 - Divide a problem in subproblems of smaller size
 - Solve the subproblems recursively (conquer)
 - Combine the solution to solve the original problem
 - When subproblems are too big to be solved directly we are in the recursive case
 - When subproblems can be solved directly we are in the base case
- The complicated part is the merging process
- It runs always as $n \cdot \log(n)$, there is not worst or best case
- It requires a lot of memory to store all the sub-arrays
- It is worse than insertion sort in the best case, but better in most cases
- It cannot work online (!)

Idea

- I want to sort the array A
- I split the array using the indices p,q,r such that $p \leq q < r$
- I want to produce a single sorted subarray
- I call initially on A with p=1 and r=A.length
- The index q is the one that best splits the array in 2
- For merging I always have sorted arrays to merge
 - The first element of each array is guaranteed to be the smallest one of the entire array
 - I compare the first element of the 2 arrays to be merged, and I put the smallest in the output array
 - I repeat until one of the arrays is empty
 - I finish by putting what remains of the other array in the output
 - I put an imaginary infinite at the end of any array
 - * This is so that when I finish the elements of an array, whatever remains in the other is smaller and so it is inserted in the output

Pseudocode

```

MERGESORT(A,p,r)
  if p < r
    q = (p+r)/2
    MERGESORT(A,p,q)
    MERGESORT(A,q+1,r)
    MERGE(A,p,q,r)

MERGE(A,p,q,r)
  n1 = q - p + 1
  n2 = r - q

```

```

for i = 1 to n1
    L[i] = A[p+i-1]
for j = 1 to n2
    R[i] = A[q+j]
L[n1+1] = \infty
R[n2+1] = \infty
i = 1
j = 1
for k = p to r
    if L[i] <= R[j]
        A[k] = L[i]
        i += 1
    else A[k] = R[j]
        j += 1

```

MERGESORT(A,1,A.lenght)

Running time

- MERGE
 - Copying the elements into the subarrays takes $\Theta(n)$
 - Adding elements to the final array takes n iterations of that themselves take constant time
* $\Theta(n)$
 - In total, the merging takes $\Theta(n)$
- MERGE-SORT
 - Let $T(n)$ be the unknown running time of MERGE-SORT
 - Calculating q: $\Theta(1)$
 - Solve recursively 2 subproblems of size $n/2$: $2T(n/2)$
 - Call to MERGE: $\Theta(n)$
 - So, $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(1) + \Theta(n) & \text{if } n > 1 \end{cases}$
 - The recursive equation can be solved and we find that $T(n) = \Theta(n \log n)$

Limiting behaviour of functions

- There are different notations to define the behaviour of functions
- The Θ (theta) notation signifies asymptotic equality
 - Formally, for a function $f(n)$ having a certain Θ notation there are 2 constant that multiplied for the Θ function are constantly greater or smaller than $f(n)$ for $n > n_0$
* This is defined as a tight bound
 - If I say that $f(n) = \Theta(g(n))$ I mean that $f(n)$ belongs to the family of functions with order of growth $g(n)$
- The O (big-O) notation indicates an upper bound for the asymptotic behaviour
 - The formal definiton is similar to that of Θ , but instead of a tight bound I only search for an upper bound
* I only want a constant, not 2 (!)
- The Ω (big-Omega) notation indicates a lower bound for the function
- An important theorem: $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$
- Factorials are faster than exponentials, but slower than n^n (!)

Recurrence equations

- A recurrence equation describes a function in terms of its value on a smaller input
- An example: analysis of a divide and conquer algorithm
 - $T(n)$ is the running time of the algorithm on input n
 - Dividing takes $D(n) = \Theta(1)$ time
 - Conquer takes the same T on a smaller input, so $aT(n/b)$
 - * I need to solve a subproblems in with an input size reduced of a factor b
 - Combining the solutions takes $C(n) = \Theta(n)$ time
 - So we have that $T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + c + cn & n > 1 \end{cases}$
- Solving recurrence equations: the iteration method
 - If I have $T(n) = T(n/2) + c \implies T(n/2) = T(n/4) + c$ and so on
 - This implies that $T(n) = c + c + T(n/4)$
 - If I continue this until I get to the base case $T(1)$
 - I can write therefore $T(n) = c * k + T(n/2^k)$
 - The base case will be when $n = 2^k$ and therefore $k = \log n$
 - So I get that $T(n) = c * \log n + T(n/2^{\log n}) = c * \log n + T(1)$
 - This means that $T(n) = \Theta(\log n)$
- Solving recurrence equations: the recursion tree method
 - I convert the equation into a tree and sum up the cost of each node
 - Let's try with $T(n) = 2T(n/2) + n^2$
 - The root has a cost n^2 and 2 children of cost $T(n/2)$ each
 - I continue to expand until the base case
 - This gives us a $O(n^2)$, since I cannot determine a tight bound

Heapsort

- Normally, the numbers to be sorted are a key that is paired to other data, forming a record
 - A record is composed of a key and satellite data
- When I want to sort, are all keys unique?
- Its running time is $\Theta(n \log n)$ and it sorts in place like insertion-sort
- It is based on a data structure called heap
 - An heap is a nearly complete binary tree
 - * All nodes are binary except for possibly the last level
 - * If the last level is not full, it is filled from left to right
 - It follows the heap property: the value of a parent must be greter that that of a children
 - * This is a max heap, there are also min heaps where the property is opposite
 - The size of an heap is the number of nodes
- We can represent an heap with an array
 - The first element is the root
 - The children of the root are the second and third element
 - The fourth and fifth element are the children of the second, and so on
 - The children of node $A[i]$ are nodes $A[2i]$ and $A[2i+1]$
 - The parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$
 - The maximum element is always the root
- We can define 3 basic functions that return the index of the left child, right child and parent of element with index i in an heap

```
LEFT(i)
    return 2i
```

```
RIGHT(i)
```

```
return 2i+1
```

```
PARENT(i)
```

```
return floor(i/2)
```

- How to maintain the max heap property (MAX-HEAPIFY)
 - I recursively explore the tree
 - If I find a parent smaller than its child, I swap them and continue
 - I assume that there is only one violation
 - The running time is $O(\log n)$, or linear to heap size ($O(h)$)

```
MAX-HEAPIFY(A,i)
```

```
l, r = LEFT(i), RIGHT(i)
```

```
if l <= A.heap-size and A[l]>A[i]
```

```
    largest = l
```

```
else largest = i
```

```
if r <= A.heap-size and A[r]>A[i]
```

```
    largest = r
```

```
else largest = i
```

```
if largest != i
```

```
    exchange A[i] and A[largest]
```

```
    MAX-HEAPIFY(A,largest)
```

- Now we start from a random array and we want to make it a max heap
 - Note that $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are leaves

```
BUIL-MAX-HEAP(A)
```

```
A.heap-size = A.lenght
```

```
for i = floor(A.lenght/2) downto 1
```

```
    MAX-HEAPIFY(A,i)
```

- This operations has an loose upper boundf of $O(n \log n)$
 - I do $n/2$ times a $O(\log n)$ operation
- However, the argument to MAX-HEAPIFY is almost never n (!)
 - In the first step it is 1, then 2 and so on
- The worst case running time of MAX-HEAPIFY is $O(\log i)$ where i is the value of the for loop in BUILD-MAX-HEAP
 - We obtain $O(n)$
- The next step is to actually sort the array
 - We swap the root with the last element and decrease the heap-size by 1
 - We call MAX-HEAPIFY on the root to rebuild the max-heap property
 - We repeat until the heap-size is 1
 - The array is sorted (!)

```
HEAPSORT(A)
```

```
BUILD-MAX-HEAP(A)
```

```
for i = A.lenght down to 1
```

```
    exchange A[1] with A[-1]
```

```
    A.heap-size = A.heap-size - 1
```

```
    MAX-HEAPIFY(A,1)
```

- The total running time is $O(n \log n)$
- Compared to mergesort, which has a $\Theta(n \log n)$, here we have an O bound
 - The worst case is like mergesort, but it can be faster (!)

Priority queues

- A priority queue is a data structure for maintaining a set S of element each with a priority value called key
- There are max and min priority queues
- A max priority queue supports the following operations
 - Return the element with largest key
 - Remove the element with largest key and return it
 - Increase the key of an element
 - Insert a new element
- Heaps are really useful for implementing priority queues
- Getting the largest element takes constant time

```
HEAP-MAXIMUM(A)
    return A[1]
```

- Extracting the largest element and re-building the heap takes $O(\log n)$ since it is essentially a single call to MAX-HEAPIFY plus a constant amount of work

```
HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heapsize]
    A.heapsize = A.heapsize - 1
    MAX-HEAPIFY(A, 1)
    return max
```

- Increasing the value of a key is $O(\log n)$ since the maximum possible number of place exchanges is equal to the height of the heap, $\log n$

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key smaller than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

- Inserting a new element into the heap is equivalent to increasing the key of an already existing one
 - I can insert an element with key infinitely small and update it to the real value
 - It uses HEAP-INCREASE-KEY so its running time is $O(\log n)$

```
MAX-HEAP-INSERT(A, key)
    A.heapsize = A.heapsize + 1
    A[A.heapsize] = - infinity
    HEAP-INCREASE-KEY(A, A.heapsize, key)
```

Quicksort

- It is a divide and conquer algorithm
- The conquer and combine parts are really easy, but divide requires effort
 - This is sharply different from MERGESORT, where combine is the most demanding part
- Divide: we want to find indexes p, q, r such that the elements $A[p \dots q - 1] \leq A[q] \leq A[q + 1 \dots r]$
 - q is determined by the PARTITION function
- Partitioning: I want to create 4 zones with indexes p, i, j, r such that $A[p \dots i] \leq A[r] < A[i + 1 \dots j]$
 - $A[r] = x$ is called pivot element and it can be chosen freely

- Usually the implementation places x at the end of the array
- The region $A[j + 1 \dots r - 1]$ is unrestricted
- The value returned by the partitioning step will be the q of the divide step
- i is initialized to $p-1$ so that there are no elements between them
- In the for loop j is always moving
 - * If the new element is bigger than x , nothing happens
 - * If it is smaller, I increase i of 1 and place it in the new i position
 - * What was previously in the i position is necessarily bigger than x , since it was in the j region, and so I place it as element j
 - * After the end of the for I have a series of elements smaller than x , a series of elements bigger, and x itself at the end
 - * I exchange x with the element $i+1$, which is necessarily bigger than it
 - * Now x is what splits the array in a smaller and a bigger subarray, and so I return its index ($i+1$ now) which will be assigned to q
- The base case is when the subarray has 3, 2, or 1 elements
 - In this case the PARTITION function puts them in order
- Conquer: the 2 subarrays $A[p : q - 1]$ and $A[q + 1 : r]$ are sorted recursively with QUICKSORT
- Combine: trivial, everything is sorted because QUICKSORT sorts in place(!)

```
QUICKSORT(A, p, r)
```

```
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

```
PARTITION(A, p, r)
```

```
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if A[j] <= x
      i = i + 1
      exchange A[i] and A[j]
  exchange A[i+1] and A[r]
  return i + 1
```

```
QUICKSORT(A, 1, A.length)
```

- The running time depends on whether the array is balanced or not
 - This in turn depends on the choice of the pivot element x
 - The choice of the pivot influences the running time (!)
 - The array is balanced if the pivot causes a constant proportional split
 - If the array is balanced we are in the best case
- The worst case running time is $\Theta(n^2)$ (unbalanced array) and the average and best case running times are $\Theta(n \log n)$
 - In practice we are almost always in the best case or close to it (!)
- The array is maximally unbalanced when it is already sorted (!)
 - In this case I have a subarray with 0 elements and one with $n-1$ elements
 - I do $n-1$ recursions, since at each step I decrease n by 1 and I stop when I get to 1
 - I pay a cost $T(0)$ for the branch with 0 elements and a cost $T(n-1)$ for the branch with $n-1$ elements
 - At each level of the tree I pay a fixed cost $\Theta(n)$ for the partitioning
 - $T(n) = T(n-1) + T(0) + \Theta(n)$
 - I have n calls of $\Theta(n)$ complexity: the cost is $\Theta(n^2)$
- The array is perfectly balanced when I get a constant proportional split
 - Doesn't matter the proportionality, it can also be a 1/100 to 99/100 split, as long as it is not 0 to $n-1$

- In this case $T(n) = 2T(n/2) + \Theta(n)$
 - $T(n)$ is of complexity $\Theta(n \log n)$
- In a random array I expect a mix of good and bad splits
 - This only adds a constant term to the complexity
 - In this case I still have a complexity of $\Theta(n \log n)$
- To assure that no particular input (i.e. a sorted array) cause the worst-case scenario of quicksort, I can use the randomized version of the algorithm
 - I can either randomize the input or the choice of the pivot

Some reflections on sorting

- All the algorithms we saw sort in place with the exception of MERGESORT
- A sort can be stable or not
 - Sorting stability is the preserving of the orders of records with equal keys
 - This is important for sorting keys with satellite data, to not mix up the satellite data order
 - * If I order column A and then column B, I see that column A is not ordered anymore even for entries with the same keys on column B
 - Insertion sort and mergesort are stable, heapsort and quicksort are not stable
- All the algorithms we saw sort by comparing elements
- The lower bound for comparison-based sorting is $\Omega(n \log n)$ because in the worst case I need to do at least $n \log n$ comparisons
 - Any comparison sort algorithm must be able to sort any possible input of size n
 - There are $n!$ possible permutations of an array of size n , and the algorithm must be able to solve all of them
 - Every permutation requires a different rearrangement of the array in order to be sorted
 - I can imagine a decision tree with $n!$ leaves, corresponding to all the permutations
 - The height of the decision tree represent the number of single comparisons that the algorithm has to do in the worst case
 - A tree of height h has 2^h leaves, and so a tree with $n!$ leaves has height $\log n!$
 - We saw before that $\Theta(n!) = \Theta(n \log n)$
- Heapsort and Mergesort are asymptotically optimal comparison sort algorithms

Counting sort

- It can sort in linear time since it is not based on comparison
- It assumes that the input array contains only integers in the range $0 - k$
- For each element x into the array, determine how many elements are equal or smaller than x
- Put x in the correct position in the array
- It uses an input array A , an output array B and a counter array C
- I first create an array C of length k containing all 0s
- I go through all the elements in A
 - In each iteration i use $A[i]$ as an index in C , and I increment that position of 1
 - In this way element $C[i]$ contains the number of occurrences of the value i in the array A
- I go through C and I add, starting from the beginning, the value of $C[i-1]$ to $C[i]$
 - In this way element $C[i]$ contains the number of elements in A that are smaller or equal to i
 - I am converting a counter for i in a cumulative counter
- Finally, I iterate j from the A .length down to 1
 - I put in the output array B the element $A[j]$ in the position that is stored in $C[A[j]]$
 - * If there are i elements in A smaller than $A[j]$, then $A[j]$ is put in $B[i]$
 - I decrease the respective counter in C of 1, so that if I find another element in A with the same key it is put in the previous position of B

COUNTING-SORT(A, B, k)

```

let C[0...k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
for i = i to k
    C[i] = C[i] + C[i-1]
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1

```

- The first and third for loops require both time $\Theta(k)$
- The second and last for loops require both $\Theta(n)$
- The total running time is $\Theta(n + k)$
- The running time is at minimum $\Omega(n)$, and $O(n+k)$
 - If $k = O(n)$, then COUNTING-SORT runs in $\Theta(n)$
 - Since I use COUNTING-SORT only when $k = O(n)$, in practice the algorithm runs in $\Theta(n)$
- COUNTING-SORT is stable
 - The last equal key element of A is the first to be put in B
 - The last for loop runs backwards, so the first element is put in the last position
 - Order is preserved
 - This is important because COUNTING-SORT is frequently used as a subroutine of RADIX-SORT
 - * RADIX-SORT requires a stable sorting subroutine

Radix sort

- It considers the key as a number in base k, which has d digit and so occupies d columns
- It looks at 1 column at a time, starting from the LEAST significant and sorting it with any stable algorithm
 - If I start from the most significant, I need to recurse for sorting the least significant digits
 - If I start from the least significant and I use a stable sort, when I sort the most significant column everything is correctly sorted
- It requires only a for loop with i from 1 to d
 - In each iteration it calls a stable sorting algorithm on column i
- It takes only d passes on the array

RADIX-SORT(A, d)

```

for i = 1 to d
    use a stable sort to sort A on digit i

```

- RADIX-SORT takes $\Theta(d * f(n))$ time, where $f(n)$ is the running time of the subroutine used
- If I use COUNTING-SORT as a subroutine, it takes $\Theta(d(n + k))$
- In decimal notation $k=9$ since a column can only hold digits in the range 0 - 9
- Therefore when the base used is smaller than n, RADIX-SORT has complexity $\Theta(n)$
- RADIX-SORT is not stable and does not sort in place
 - If memory is a problem then QUICKSORT is better

Dynamic sets

- In CS, differently from maths, sets can change
- A set supporting elementary operations is a dictionary
 - Elementary set operations are insertion, deletion and test membership
- Each set element has key, and optional features
 - It can have satellite data

- It can have a pointer, which points to another element in the set
- Set operations can be queries or modifying operations
 - A query always returns a pointer to an element in the set
 - A modifying operation modifies the set
 - * INSERT(S,x) and DELETE(S,x)
- Dynamic sets can be represented with different data structures: stacks

Stack

- It is a pile of elements on top of each other
- A new element is always added to the top of the stack: PUSH(S, x)
- Elements are always removed from the top of the stack: POP(S)
- Popping order is the reverse of the push order
 - They follow the last in first out (LIFO) policy
- A stack is NOT a good data structure for sorting and it is not used for this purpose
- Some applications of stacks
 - Storing undo history in text editors
 - Synthax parsing: evaluating missing parentheses
 - * I push open and close parenthesis to the stack and pop twice when I find matching parenthesis
 - * At the end of the file I require the stack to be empty
- A stack of n elements can be implemented with an array S[1..n]
- The S.top call returns the index of the top of the stack
- STACK-EMPTY(S) and STACK-FULL(S) return true or false in O(1)

STACK-EMPTY(S)

```
if S.top == 0
    return True
else
    return False
```

STACK-FULL(S)

```
if S.top == S.lenght
    return True
else
    return False
```

- PUSH(S,x) is also O(1)
- If the size of the stack is not infinite I need first to check if it is full

PUSH(S,x)

```
if not STACK-FULL(S)
    S.top = S-top + 1
    S[S.top] = x
else
    error "stack is full, cannot push"
```

- POP(S) returns the element we popped and removes it from the stack

POP(S)

```
if not STACK-EMPTY(S)
    S.top = S.top - 1
    return S[S.top + 1]

else:
    error "stack empty, nothing to pop"
```

Queue

- In a queue I use a FIFO policy instead of a LIFO
- I add elements to the queue with the enqueue operation in $O(1)$
 - New elements are added at the end of the queue
- Elements are removed with the dequeue operation
 - They are always removed from the top of the queue
- Also queues can be implemented with arrays
- Queues are circular, there is no end and beginning for the array (!)
 - For n elements I need an array of $n+1$ size (!)
 - This is because I need an empty element for marking the end of the queue
- I define several attributes for the queue
 - `Q.head` is the index of first element of the queue
 - `Q.tail` is the index of the last element of the queue + 1, it is the next available position
 - * It points to an empty element of the array (!)
- Initially I have that `Q.head = Q.tail = 1`
- The queue is full when `Q.head = Q.tail + 1` (circular case) or when `Q.head = 1` and `Q.tail = Q.lenght` (linear case)

QUEUE-EMPTY(Q)

```
if Q.head == Q.tail
    return True
else
    return False
```

QUEUE-FULL(Q)

```
if Q.head == Q.tail + 1 or (Q.head == 1 and Q.tail = Q.lenght)
    return True
else
    return False
```

ENQUEUE(Q,x)

```
if QUEUE-FULL(Q)
    error "Queue is full, cannot add"
Q[Q.tail] = x
if Q.tail == Q.lenght
    Q.tail = 1
else
    Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
if QUEUE-EMPTY(Q)
    error "Queue is empty, nothing to dequeue"
x = Q[Q.head]
if Q.head == Q.lenght
    Q.head = 1
else
    Q.head = Q.head + 1
return x
```

Arrays

- They are easy and fast to use, they can implement many data structures
- It is really inflexible for organising data

- I need (directly or not) to specify the size of the array at the beginning
- I cannot implement all data structures with arrays

Linked lists

- It is like an array where elements are next to each other
- The order is NOT determined by indices, but by pointers in each element for the next element
- They are allocated dynamically when new elements are added
- L.head is a pointer to the first element of the linked list
- Each element x has 2 attributes
 - x.key is the content of the element
 - x.next is a pointer to the next element
- If x.next = NIL it means that we reached the end of the list
- If x.head = NIL the list is empty
- In double linked lists each element has also a x.prev attribute
 - x.prev is a pointer to the previous element in the list
- Returning an element is $O(1)$ in an array but $O(n)$ in a linked list
 - I need to go through all the elements (!)

SEARCH-LIST(L, k)

```
x = L.head
while x != NIL and x.key != k
    x = x.next
return x
```

- Inserting at the beginning of a double linked list is $O(1)$

LIST-INSERT(L, x)

```
x.next = L.head
if x.head != NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

- Deleting when I have a pointer x is general and it takes also $O(1)$

LIST-DELETE(L, x)

```
if x.prev != NIL
    x.prev.next = x.next
else
    L.head = x.next
if x.next != NIL
    x.next.prev = x.prev
```