Algorithms and Data Structures

Summer School 2019, Singapore

Prof. Dr. Christian Pape

Contents

Contents

Content

Literature

- No specific book.
- Standard: Introduction to Algorithm. MIT Press. T. Cormen, C. Leiserson, R. Rivest, and C. Stein.
- References at the end of slides.

Content

Algorithm
Definition

Definition (Algorithm)

An **algorithm** is a well formed computable process producing one ore more output values for one ore more input values.

"'Wenn CD nicht Teiler von AB ist, subtrahiert man, von den beiden Zahlen AB und CD ausgehend, immer die kleinere von der größeren bis die entstandene Zahl Teiler der ihr vorhergehenden ist"'[Euklid: Die Elemente, Buch VII.2]
If CD is not a divisor of AB, subtract, beginning from AB and CD, the smaller from the greater number until the result is a divisor of the former one.

Example (Euclid's Algorithm)

Input: Two integers a und b
Ouput: Greatest common divisor of a und b
While a and b are not equal, subtract the smaller number from the greater.

Algorithm

Pseudocode

Our given algorithm is not precise and therefore not well formed or computable. Euclid's version is more precise.

- Value range of input numbers *a* and *b*? Negative? Zero allowed?
- Computable steps are not formulated precise enough.
- What is the produced output?
- We us **pseudocode** for a precise formulation of algorithms.
- Pure mathematical representation possible, but not readable for humans anymore.

Listing 1: Euklid's Algorithm

```
ggt = Euklid(a,b)
input: $a, b \in \mathbb{N}$
output: greatest common divisor $ggt$ of $a$ and $b$
while a $\neq$ b
  if a > b
    a $\leftarrow$ (a - b)
  else
    b $\leftarrow$ (b - a)
ggt $\leftarrow$ a
```

Algorithm
Pseudo Code

- Procedural
- Blocks build by indentation (shorter programs)
- Basis data types from math like $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \ldots$. Often omitted.
- Input parameters given in parenthesis after function name.
- Output variable(s) declared before $=$ in the function declaration
- Usual control statements like if-else, while, for, do-while
- (Multiple) Assignment $\leftarrow$
- For loop: the lower and upper bound value u of the loop variable i is given. After the loop i still exists. Its value after the loop is $u + 1$!
- for i $=$ 1 to 10 ...
- End line comments \\ a comment

Algorithm

Pseudo Code

Listing 2: Minimum Search

```
min = minimum−search(a)
**input**: array a $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ with $n \geq 1$
**output**: $\min\{a_0, \ldots, a_{n-1}\}$
min ← a[0]
i ← 1
while i < a.length
  if a[i] < min
    min ← a[i]
    i ← i + 1
```

Listing 3: Minimum Search with for

```
min = minimum−search2(a)
min ← a[0]
for i ← 1 to a.length
  if a[i] < min
    min ← a[i]
// i is a.length + 1 at this line
```

Algorithm
Problem

Definition (Problem)

A **problem** is a well defined description of the expected output for a defined input.

Example (Greatest Common Divisor)

Input: $a, b \in \mathbb{N}$
Output: Greatest common divisor a und b

Definition (Instance)

An **instance** of a problem is one example of the input (and sometimes expected output) described by the problem.

Example (Greatest Common Divisor)

$a = 9, b = 15$ is an instance of the above problem
$a = -5, b = 11$ not

Algorithm

Correctness

Definition (Correctness)

An algorithm is **correct** for a given problem, if it **terminates** for each instance of the problem and produces the defined output for the instance.
A correct algorithm **solves** the problems.

Definition taken from [3]. Sometimes termination is not part of the correctness definition: partial correctness. If it terminates: total correctness.

Example

Euklid(a,b) is correct

- Proof necessary.
- Correctness can not be proven with tests!
- Non trivial problems have a infinite number of instances.
- Proofs are not part of this lecture. You have to trust me.

Content

Time Complexity

- Pseudo code can not be execute.
- Execution time can not be measured but *estimated*
- Goal: Find the *best* algorithm for a problem
- Estimation of number of steps (basic statements) an algorithms executes.
- We are interested on problem instances where the algorithm performs very well (base case), is very slow (worst case).
- Additional: average performance
- Problem: Infinite number of problem instances exists.
- Solution: Estimate performance for instances with a given input **complexity**.
- Complexity n is defined for a problem not for an algorithm.
- n usually is given. If the instance is large or complex, n should be large, and vice versa.
- Other resources like used memory space can be estimated in the same manner.

Time Complexity

Example Best Case

- Input complexity of greatest common divisor $n := \max\{a, b\}$
- Best case is $a = b = n$ because loop terminates immediately.
- We have to "execute" the algorithms for all inputs of length n and count basic statements. Here: Each executed operator.

Listing 4: Euklid's Algorithm

```
ggt = Euklid(a,b)
while a ≠ b
  if a > b
    a ← (a − b)
  else
    b ← (b − a)
ggt ← a
```

Operator	Frequency
$a \neq b$	1
$a > b$	
$a - b$	
$a \leftarrow \dots$	
$b - a$	
$b \leftarrow \dots$	
$ggt \leftarrow a$	1

Result is given as a function in n: $T_{bc}(n) = 2$

Time Complexity

Example Worst Case

- Worst case is $a = n, b = 1$ because a is decreasing very slowly.

Listing 5: Euklid's Algorithm

```
ggt = Euklid(a,b)
while a ≠ b
  if a > b
    a ← (a − b)
  else
    b ← (b − a)
ggt ← a
```

Operator	Frequency
a ≠ b	n
a > b	$n-1$
a − b	$n-1$
a ← ...	$n-1$
b − a	
b ← ...	
ggt ← a	1

Result: $T_{wc}(n) = 4n - 2$

Time Complexity

Simplification

- Estimation is inaccurate therefore the constant factors like 4 or 2 in $4n - 2$ are mostly irrelevant.
- In practice every program can be accelerated by a constant by better hardware or compiler.
- We tolerate inaccuracies in the magnitued of a constant factor.
- Worst case of example is $T_{wc}(n) = c_1 \cdot n + c_2$ for constants $c_1, c_2 \in \mathbb{Z}$.
- Time complexity in example is linear in n.
- Only dominant factor of result is important.
- We give and compare the results of algorithms within the Big O notation.
- $T_{wc}(n) = O(c_1 \cdot n + c_2) = O(n)$
- As a consequence we only need to count the statements which are execute most, e.g. loop condition $a \neq b$ because the if-else only executes a constant number of operations.

Big O Notation
Definition

- f and g are function from $\mathbb{N}$ onto $\mathbb{R}$

Notation	Definition	Intuition	Analogy
$f \in O(g)$ $f(n) = O(g(n))$	$\exists c > 0, \exists n_0 \in \mathbb{N}$ $\forall n \geq n_0 : f(n) \leq c\|g(n)\|$	f grows no faster than g	"$f \leq g$"
$f \in \Theta(g)$	$f \in O(g)$ und $f \in \Omega(g)$	f grows equally to g	"$f = g$"
$f \in \Omega(g)$	$g \in O(f)$ (Knuth)	f faster then g	"$f \geq g$"

- O and Ω also includes equality.
- We want to estimate as best as possible with Θ
- Previous example hold for Θ (and therefore Ω) as well

Big O Notation
Rules for Simplification

- $f \in \Theta(f)$
- $\Theta(c \cdot f(n)) = \Theta(f(n))$
- $\Theta(f(n) + g(n)) = \Theta(f(n)$ if f grows faster than g $(f \in \Omega(g))$
- These rules can be proven for O, Ω, and Θ by their definitions.
- Functions should be simplified as much as possible with these rules.

Example

- $T(n) = 4n - 2 = \Theta(4n - 2) = \Theta(4n) = \Theta(n)$
- $T(n) = 2^{n+1} + n^8 = \Theta(2^{n+1} + n^8) = \Theta(2^{n+1}) = \Theta(2 \cdot 2^n) = \Theta(2^n)$

Big O Notation

Rules for Simplification

Example

Simplify the following functions with the Θ notation.

1. $T(n) = 2n^3 + 10n^2 - 5$
2. $T(n) = (n+7)^3$
3. $T(n) = 1 + 2 + \ldots + n - 1 + n = \sum_{i=1}^{n}$
4. $T(n) = 2\log_2 n + 4\log_4 n$ (Use $\log_b x = \frac{\log_a x}{\log_a b}$)
5. $T(n) = n\log_2 n + 2n\log_2 \frac{n}{2}$ (logarithmic rules?)

Example

Difficult: Show that $\log_2(n!) = O(n\log_2 n)$ (Try to make $n!$ larger so simplify with logarithmic rules)

Prime Sieve

Algorithm

Problem

Input: *A natural number* $n > 2$
Output: *All prime numbers from* 2 *up to* n

Idea: Write down all numbers from 2 to n in a (long) line (on the beach during low tide) Start with the first prime 2 and cross out all multiples of 2. The next non crossed out number is a prime. Continue until n is reached (or the water comes back).

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	3		5		7				11		13				17		19				23		25
2	3		5		7				11		13				17		19				23		

Possible Optimization: Stop when $\lceil \sqrt{n} \rceil$ is reached

Prime Sieve
Algorithm

- Implementation with a boolean array: index specifies the number, sieve[i] = false means i is crossed out.

Listing 6: Prime Sieve

```
prime-sieve(n, sieve)
input: natural number $n > 2$ and reference to a boolean
array sieve with length $n+1$
output: sieve (with changed values)
        such that $sieve[p] = true$ iff p is a prime for all $p \leq 2$
for i ← 0 to sieve.length
  sieve[i] = true
for p ← 2 to $n$
  if sieve[p] = true
    for i ← 2 * p to $n$ step $p$
      sieve[p] ← false
```

Prime Sieve
Algorithm

1. Execute the algorithm for $n = 25$ (on paper). Write down the resulting array after each inner for loop.
2. Best, worst, and average case are the same, since there is only one instance of the problem for a given n.
3. Give a good estimation of $T(n)$. You only need to count the number of executions of the inner if statement. Do not try giving a closed expression. Use something like $a + b + \ldots + z$.
4. Easy: Show $T(n) = O(n^2)$
5. Harder: Show $T(n) = O(n \log n)$ (natural logarithm, try something with the harmonic series).
6. Very hard (don't try it): $T(n) = \Theta(n \log \log n)$ (a result from number theory is needed).

Contents

Content

Definition

Problem (Selection Problem, k-smallest Element)

Input: *A sequence a of n values, $1 \leq k \leq n$.*
Output: *Value $x \in a$ that is larger than exactly $k-1$ other values of a.*

A total ordering $\leq$ on the values of a is necessary, like $\leq$ for numbers.
Consequence of definition: x is the k-th value in the sorted sequence of a

Example

$a = \{9, 6, 4, 1, 8, 2, 5, 6, 4, 3\}, k = 5$
4 is the 5th-smallest element of a
1, 2, 3, 4, $\underline{4}$, 5, 6, 6, 8, 9

Important special cases:

- $k = 1$: Minimum
- $k = n$: Maximum
- $k = \lfloor \frac{n}{2} \rfloor$: Median

Applications

Example (Statistics)

Given are all house hold incomes from a country.
House holds belong to the **upper class** if their income is larger than 60% of the median income.

Example (Tournaments)

Given are n soccer teams. Which ones are the top three.
Total ordering is a problem: How and how often have teams to play against each others. Assume there is tournaments modus that results in a total ordering.
Top three teams are $n-2, n-1$, and n-smallest teams with respect to this ordering.

Naive Algorithm

- Find the minimum of the sequence.
- If $k = 1$ we are done.
- If not: remove the minimum. And try again with $k - 1$ (recusive or loop)
- Removing in an array to complex: swap minimum to the front and continue with rest of array.

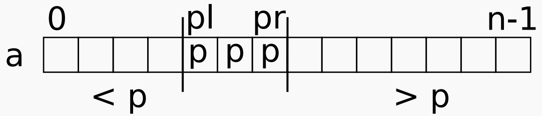
Listing 7: Naive Selection

```
element = selection-naive(a, k)
for i ← 0 to k−1
  j ← i
  for l ← i + 1 to n
    if a[l] < a[j]
      j ← l
  a[i], a[j] ← a[j], a[i]
element ← a[k − 1]
```

- Problem complexity is n
- Best case is $k = 1$: $\Theta(n)$
- Worst case is $k = n$: $1 + 2 + 3 + \ldots + (n - 1) = \frac{n(n-1)}{2} = \Theta(n^2)$
- Average case: $\Theta(n^2)$

Better Algorithm

- Try a divide and conquer approach.
- Find a pre-processing such that afterwards for some $p \in a$ (pivot) the following condition (invariant) holds.



- Known as **three-way-partitioning**.
- Afterwards only three cases exist:
  1. $pl \leq k-1 \leq pr$: p is the k-smallest element.
  2. $k-1 < pl$: k-smallest element is in the left hand part
  3. $k-1 > pr$: k-smallest element is in the right hand part

Better Algorithm

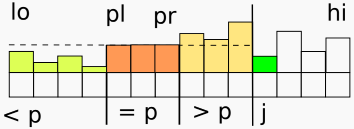
- In case 2 and 3 the algorithm has to be called on smaller parts of *a*.
- Pre-processing must be correct for $a[lo..hi]$ with $0 \leq lo \leq hi \leq n-1$.
- Assume a function `(pl, pr, p) = partitioning(a, lo, hi)` that does the desired pre-processing by reordering the values within *a*.
- The following recursive algorithm then solves our problem.

Listing 8: Selection

```
element = select(a, k, hi, lo)
pl, pr, p ← partitioning(a, lo, hi)
if pl ≤ k − 1 and k − 1 ≤ pr
  element ← p
else if k − 1 < pl
  element ← select(a, k, lo, pl − 1)
else
  element ← select(a, k, pr + 1, hi)
```

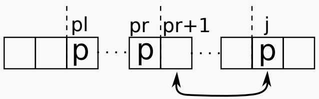
Better Algorithm

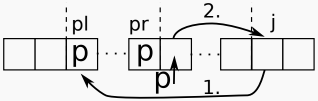
- Three way partitioning from Edsgar W. Dijkstra [4]
- Extend a[lo..j-1] by a[j]
- Three cases exists



Listing 9: Three way partitioning

```
(pl, pr, p) = partitioning(a, lo, hi)
p, pl, pr ← a[lo], lo, lo
for j ← lo + 1 to hi
  if a[j] = p
    a[j], a[pr+1] ← a[pr+1], p
    pr ← pr + 1
  else if a[j] < p
    a[pl], a[j], a[pr+1] ← a[j], a[pr+1], p
    pl, pr ← pl + 1, pr + 1
```





Better Algorithm

Example

Example

$a = \langle 4, 8, 5, 2, 4, 1, 3 \rangle$
Execute partitioning(a, 0, 6)

Example

$a = \langle 4, 8, 5, 2, 4, 1, 3 \rangle$
Execute select(a, 6, 0, 6).
After each call of partitioning write down the resulting a.

Better Algorithm

Analysis

- Time complexity for partitioning is linear.
- Best case select: a[0] is the k-smallest element, linear time as well
- Worst case: recursive call into a "half" with n-1 elements. This is the case when $k = n$ and a is sorted. $n + (n-1) + \ldots + 2 + 1 = \Theta(n^2)$
- Average case is $\Theta(n)$ but analysis is difficult.
- Algorithm can be further improved with better selection of p: Worst case $\Theta(n)$. Median-of-median algorithm.
- No algorithm can perform better then $\Theta(n)$ in the worst case to solve the selection problem since each elements has to be inspected at least once.

Content

Definition

Problem (Sorting, in-place)

Input: *Sequence a of values and a total ordering $\leq$ on these values.*
Output: *A reordering a such that a is sorted in ascending order (and still contains the original values of the input).*

Example (Sorting numbers)

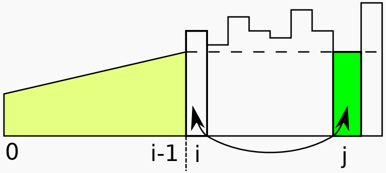
$a = \langle 9, 6, 4, 1, 8, 2, 5, 6, 4, 3 \rangle$ with the usual ordering on natural numbers
Sorted sequence: $1, 2, 3, 4, 4, 5, 6, 6, 8, 9$

Selection Sort

- Naive select algorithms already sorts the first k elements.
- Selection sort is identical to the naive select algorithm for $k = n$
- Complexity $\Theta(n^2)$

Listing 10: Three way partitioning

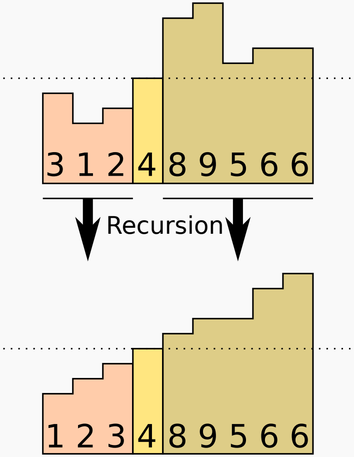
```
selection-sort(a)
for i ← 0 to n − 1
  j ← i
  for l ← i + 1 to n
    if a[l] < a[j]
      j ← l
  a[i], a[j] ← a[j], a[i]
```



Quicksort

Idea

- Again we try a divide and conquer approach.
- We use the existing three-way-partitioning.
- Pivot element p is already at is correct position after the partitioning.
- Since all values in the left hand part are less than p and all values in the right hand part are larger, sorting both parts solves the sorting problem.
- We solve both parts recursively.
- We terminate the recursion if there is only one or none element to sort.



Quicksort

Listing 11: Quicksort

```
quicksort(a, lo, hi)
if lo < hi
  p, ql, qr ← partitioning(a, lo, hi)
  quicksort(a, lo, ql − 1)
  quicksort(a, qr + 1, hi)
```

- Named Quicksort because of its good practical performance.
- Invented by Anthony Charles Richard Hoare [6] 1961.
- He used a different, two-way-partitioning, that uses less swaps.
- A improved three-way-partitioning was developed by Robert Sedgewick.
- Quicksort became the sorting routine in the C program library.

Quicksort

Example

Example

Sort the array $a = \langle 4, 9, 2, 3, 6, 8, 1, 6 \rangle$ with Quicksort.
Draw the call tree for all recursive calls.

Quicksort

Analysis

- Worst case: a already is sorted and p is the maximum of a[lo..hi]. The "half" to the right is empty. We always recurse into the large left half.

$$T(n) = (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$$

- Best case: p always is the median of a[lo..hi]. The time complexity can be express by the following **recurrence relation**

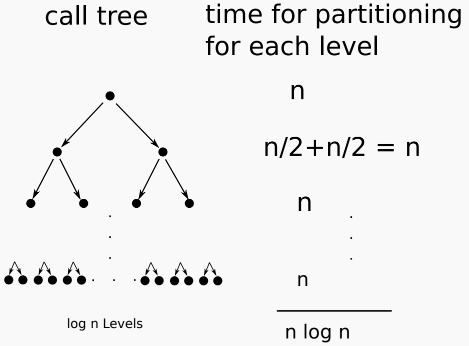
$$T(n) = 2 \cdot T(\frac{n}{2}) + n$$

- Average case (without proof):

$$T(n) = \Theta(n \cdot \log n)$$

Sorting
Quicksort

- Guessing the solution with the help of the **call tree**
- Each node is a call to quicksort. Recursive calls ar drawn with an arrow.



- $T(n) = \Theta(n \log n)$
- The result can be proven by induction on n using the definition of Big O.

Contents

Data Type

Definition

Definition (Data Type)

A **data type** is a set of values and operations on these values.

Example

In math: Set $\mathbb{R}$ of real numbers.
Operations on this set are, for instance, sum $+$, equality $=$ or the relational operator $<$.

Example

In Java: data type **int** consists of all integral numbers in the range of $\{-2^{31}, \ldots, 2^{31} - 1\}$.
There operations like binary sum $+$, identity $==$ or relational operatore $<$.

- Operations can be given or implemented as functions or methods.

Abstract Data Type

Definition

Definition (Abstract Data Type)

An **abstract data type** is a data type, with no implementation or encoding of values and operations.

Example

$\mathbb{R}$ as used in school math.
This set never really was defined (its difficult).
You learned these numbers by example (inductively) and just trusted your teacher.

- Specific encoding of Java int: 8-bit two-complement.
- In Java, Go, or C++ abstract data types are given as interfaces or abstract classes.
- Human readable description for methods are used to completely describe the behavior a the data type.

Data Structure

Definition

Definition (Data Structure)

A **data structure** is a data type, that allows to store and organize data.

Example

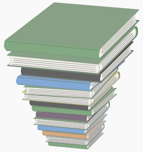
Every programming language already defines basic data types like integers or boolean.
With objects, records, structs, or union the programmer can create new compound data types.
In Java, Go, or C++ a string is a data structure.

Example

In math sets and n-tuple serve the same purpose.

Content

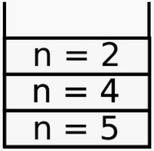
Definition

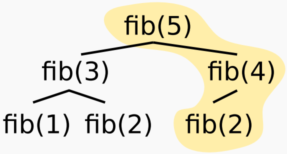

Open-Clip-Art. `bookstack.svg` von J_Alves

- A **stack** organize values from bottom top.
- A value can be pushed on top of the stack or the top value can be removed from it.
- A stack can be empty.

Example

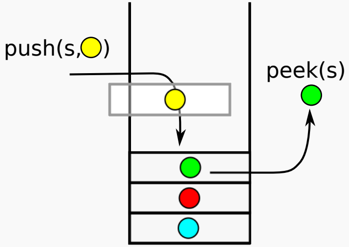
Call stack in programming languages to organize data local to a function call.





```
fib = fib(n)
fib ← 1
if n > 2
  fib ← fib(n−2) + fib(n−1)
```

Operations



- empty(s): Returns true if the the stack s is empty; false otherwise.
- push(s, x): Moves x on top of the stack s.
- peek(s): Returns the top of the stack, without removing it.
- pop(s): Removes the top from the stack and returns it.

- No error handling is given in this example.
- The four methods together with their full description represent an **abstract data type**.
- Java and Go support the definition of abstract data types with interfaces, C++ with abstract classes and virtual methods.

Implementation

- The stack s consists of an array s.stack and an integer s.top.
- s.stack[s.top - 1] is the top value of s. If $s.top = 0$ then the stack is empty.
- No specific array type is given.

Listing 12: Empty

```
empty = empty(s)
empty ← s.top = 0
```

Listing 13: Peek

```
x = peek(s)
x  ← s.stack[s.top − 1]
```

Listing 14: Push

```
push(s,x)
s.stack[s.top] ← x
s.top ← s.top + 1
```

Listing 15: Pop

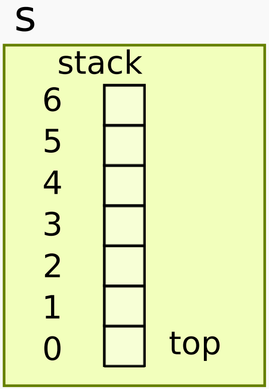
```
x = pop(s)
x ← peek(s)
s.top ← s.top − 1
```

Example

Example

The initial stack s is empty. s.stack should have length 7.
Apply the following sequence of operations on s.

1. push(s, 3)
2. push(s, 5)
3. push(s, 2)
4. pop()
5. push(s, 1)
6. pop()
7. push(s, 7)
8. pop()
9. x = peek()



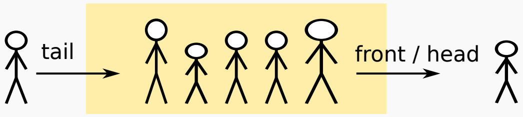
x = ??

Content

Definition

Definition

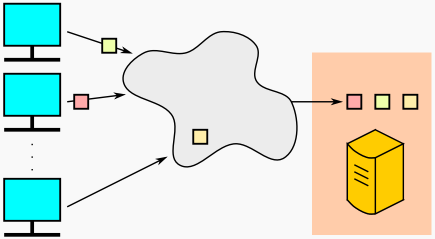
A **queue** contains a sequence of values. It allows one value to be inserted at the end (tail) of the queue, and removal of the head element.
This behavior is known as First-In First-Out (FIFO) principle.



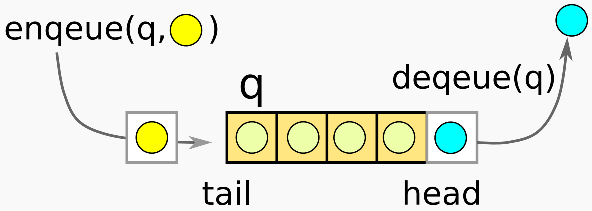
- Stack: Last-In First-Out (LIFO) principle.

Applications

- Web-Server: incoming web request (from one client) has to processed in sequenced.



- Message-Queue-Systems: Sending and receiving messages while preserving their sequential order.
- Shop-System: If items are only available with limited supply, the first shopper gets it (not the last or random).



- enqueue(q, x): Inserts value x at the tail of the queue q.
- dequeue(q): Removes the front element from the q and returns it.
- Queues are typically implemented with lists or arrays (no pseudo code given).

Content

Definition and Operations

- A **priority queue** (or heap) allows to store values x with an addition priority value x.prio.
- Only the value with the smallest priority can be removed.

Some of the essential operations:

- `priority-enqueue(q, x)`: Inserts x into the priority queue q.
- `priority-dequeue(q)`: Removes and returns the value with minimum priority from q.
- `priority-decrease-value(q, x, w)`: Reduce the priority of the value x to w. x is an element of the queue.

Reversing the order results in a maximum heap.

Applications

Example

Process management of an operating system.
The process with the highest priority is selected for execution of (a core) in the CPU.
Process with lesser priority have to wait.

Example

The sorting algorithm heap sort is based on the idea of selection sort. But it uses a minimum heap to retrieve the minimum efficiently.

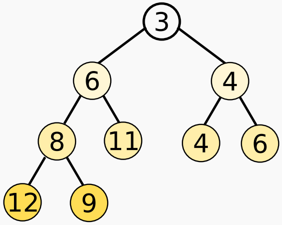
Example

Several graph algorithms use a priority queue to store data (example later).

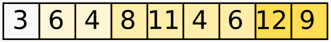
Binary Heap
Definition

Definition

A **binary (minimum) heap** is a binary tree such that for each node in the tree, the value or key of the node is smaller the all values in child nodes.



- Consequence: The minimum is always stored in the root of the tree.
- Restriction: We only consider heaps where all levels except the bottom most are complete. The bottom level must be filled with values from the left to the right (without gaps).



3	6	4	8	11	4	6	12	9

- Representation with a array: all values are stored in level order.

Binary Heap

Example

The following integer values are given:

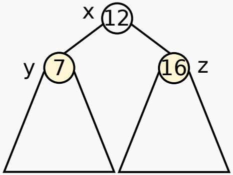
$$5, 3, 7, 2, 11, 9, 2, 8, 4$$

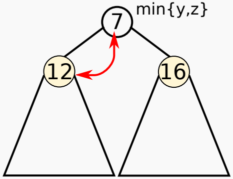
Give a binary minimum heap represented as a binary tree and with an array. Both should represent the same heap.

Binary Heap

Building a binary heap

- Building a binary heap from n given values.
- Idea: One value itself represents a heap. Try to merge two heaps of the same size, the left with top value y and the right with z with one addition value x
- Two cases:
  1. $x = \min\{x, y, z\}$: Make x the root results in a new heap.
  2. Swap x with $\min\{y, z\}$. Continue (recursively) in the corresponding sub tree. The result is a binary tree again.
- x **sinks down** the heap.





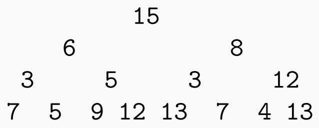
Binary Heap

Building a binary heap

Example

- Eight heaps are given consisting of the single values $7, 5, 9, 12, 3, 7, 4$, and 13.
- Merge these heaps with four other values $3, 5, 3$, and 12
- Merge the four resulting heaps with the values 6 and 8.
- Merge both heaps with 15 .

The method builds the tree bottom up in *linear* time (without proof).



Binary Heap

Building a binary heap

```
           15
    6               8
 3     5      3       4
7  5  9 12 13  7 12 13
```

```
           15
    3               3
 5     5      7       4
7  6  9 12 13  8 12 13
```

```
           3
   5               3
 6     5      7       4
7 15  9 12 13  8 12 13
```

Binary Heap

Building a binary heap

This method also is applicable if the bottom level is not complete.

```
            2
      8           4
   6     9     3     12
  7  3  5
```

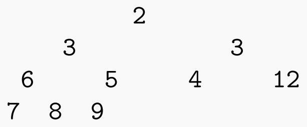
```
            2
      8           4
   3     5     3     12
  7  6  9
```

```
            2
      3           3
   6     5     4     12
  7  8  9
```

Binary Heap

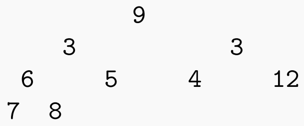
Removing the minimum

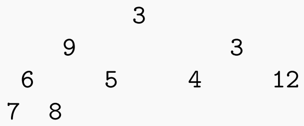
- 1. Remove top element and put the last element on top (or simply swap both).
  2. Let the new top element sink down.
- Example: remove minimum element 2.

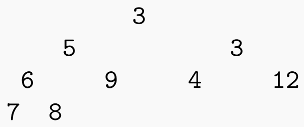


Binary Heap

Removing the minimum





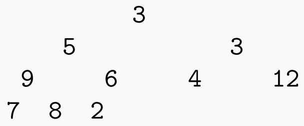


Binary Heap

Insering a value

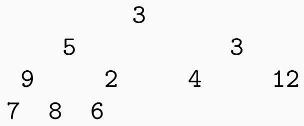
Inserting a new value x into the heap.

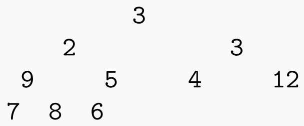
1. x is placed at the end of the heap. If the last level is full, it is placed on the front of the next level.
2. x is **raised** to the top: Compare x with its parent node y. If $x < y$ hold, then swap x with y vertauschen. Repeat until $x \geq y$ holds or x is the new root.

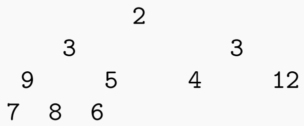


Binary Heap

Insering a value







Binary Heap
Analysis

Complexity $n :=$ number of values in the heap.
Worst Case:

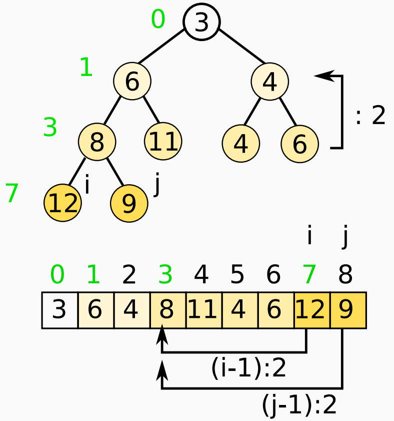
- Let M be a set of n values, `priority-queue-create(q, M)`: $\Theta(n)$
- Sinking and raising an element per level at most one swap operation.
- `priority-enqueue(q, x)`: $\Theta(\log n)$
- `priority-dequeue(q)`: $\Theta(\log n)$
- `priority-decrease-value(q, x, w)`, Remove x, change its priority and insert it again: $\Theta(\log n)$

Best case: $\Theta(1)$ except `priority-queue-create(q, M)`.

Binary Heap

Implementation

- Heap is represented with an array $a[0..n-1]$
- Move one level up: divide index by 2 (only half values on the level above)
- Parent of $a[i]$ is $a[\lfloor \frac{i-1}{2} \rfloor]$
- Left child of $a[i]$ is $a[2 \cdot i + 1]$, right child $a[2 \cdot i + 2]$
- Implement sinking and raising with a loop.



Binary Heap

Implementation

- Heap q contains an attribute $q.n$ storing the number of values in the heap and the array $q.heap$. We assume that the array is large enough to store all values.
- Let value $a[0]$ sink into the heap: `down-heap(q)`.
- Raising x in the heap: `up-heap(q,x)`.

Listing 16: Let top sink into the heap

```
down-heap(q)
i ← 0
do
    left ← 2 * i + 1
    right ← 2 * i + 2
    minimum ← i
    if left < q.n and q.heap[left] < q.heap[minimum]
        minimum ← left
    if right < q.n and q.heap[right] < q.heap[minimum]
        minimum ← right
    if minimum ≠ i
        swap(q.heap[i], q.heap[minimum])
    i ← minimum
while i < q.n − 1 and i < minimum
```

Binary Heap

Implementation

Listing 17: Raise element x to the top

```
up-heap(q, x)
i ← q.n
q.heap[i] ← x
while i ≠ 0 and q.heap[(i−1)/2] > x
  swap(a[(i−1)/2], a[i])
  i ← (i−1)/2
```

- Implementing the operations of a priority queue is left as an exercise.

Binary Heap

Example

- Build a binary heap bottom up from the values $15, 14, 13, 12, 11, \ldots, 2$.
- Add 1 to the heap.
- Remove top (1 again)
- Use the tree representation (15 is top value, 2 is the last one)

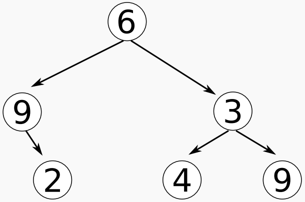
Contents

Content

Definition

Definition

A **binary tree** is either empty or consists of a node and a left and a right subtree. The node can store additional data like a key and associated values.



A binary tree can be build with compound data types, for instance a struct (Go example)

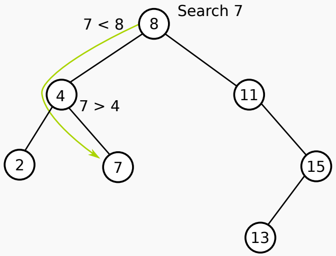
```
typedef struct Node {
  key int
  left, right *Node
}
```

Definition

Definition (Binary search tree property)

For each node x of a binary search tree with key $x.key$ the following properties hold:

- All search keys of the left subtree $x.left$ are less than $x.key$ and
- all search keys of the right subtree $x.right$ are larger than $x.key$.

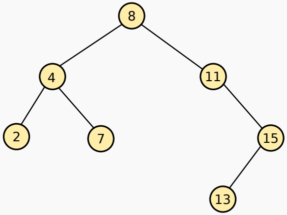


Searching

- We use the recursive structure of a binary search tree for a recursive algorithm:

Listing 18: Searching in a binary search tree

```
y = binary-tree-search(x, key)
input: a node x of the tree, a search key
output: node y with
y.key = key or null, if no such node exists
if x = null or x.key = key
  y ← x
else if key < x.key
  y ← tree-search(x.left, key)
else
  y ← tree-search(x.right, key)
```



Example

Search subsequently for 7, 3, and finally for 15 in the given example tree.

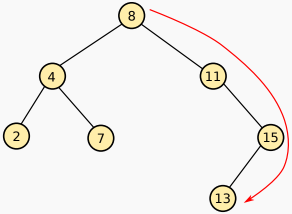
Analysis

- We count the number of comparisons done while searching.
- Base case: the algorithm stops at the first node (root); no recursive calls.

$$\Theta(1) \text{ comparisions}$$

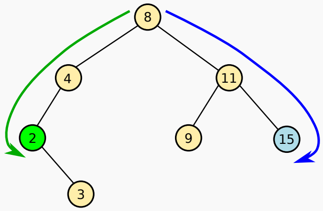
- Worst case: The tree consists of n node, each node is the left node of its parent (the tree is a linear list)

$$\Theta(h) = \Theta(n) \text{ comparisions}$$



Minimum and Maximums Search

- Minimum (iterative): Follow all left sub trees from root until no left sub tree exists. The key of resulting node is the minimum.
- Minimum (recursive): The minimum of a binary search tree is
    - either the key of the current node if *node.left* = *null*
    - or the minimum of *node.left*
- Analogous maximum: follow the right subtrees.



Listing 19: Iterative minimum search

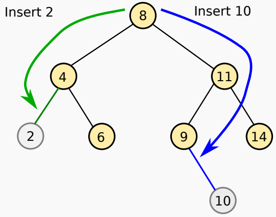
```
y = binary-tree-minimum(x)
while x ≠ null and x.left ≠ null
  x ← x.left
  y ← x
```

Listing 20: Recursive minimum search

```
y = binary-tree-minimum(x)
if x = null or x.left = null
  y ← x
else
  y ← binary-tree-minimum(x.left)
```

Inserting

- Simplification: We do not allow insertion of keys that are already in the tree.
- We start searching for a node with the key, we want to insert.
- If the key is already contained in the tree, then we stop.
- If not, then the search ends at a leaf; the new node is inserted as a new left or right sub tree for this leave.



Insert

We store the root node of the tree in a new data structure T with $T.root$.

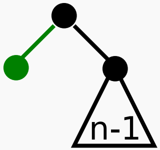
Listing 21: Iterative Insert

```
tree-insert(T, z)
input: Binary search tree $T$ and a new node $z$
y ← null
x ← T.root
while x ≠ null
  if z.key = x.key
    error "double key"
  y ← x
  if z.key < x.key
    x ← x.left
  else
    x ← x.right
  if y = null
    T.root ← z
  else if z.key < y.key
    y.left ← z
  else
    y.right ← z
```

Insert

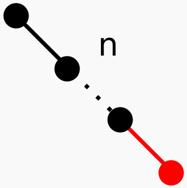
Analysis

- Counting number of comparisons between keys.



- Best case: No left sub tree exists for the root. Right sub tree contains $n-1$ node. A node with minimum key value is inserted to the left.

$$\Theta(1)$$



- Worst case: The tree degenerates to a linear list with n node. For instance to the right. A node with the Maximum is inserted.

Insert

Average Case

- For inserting, searching, minimum or maximum search (and deleletion as well)

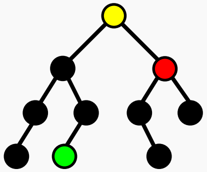
Theorem

A binary search tree created from inserting randomly distributed keys has a height of $O(\log_2 n)$.

- Consequence: All above operation use $O(\log n)$ comparisons in the average case.
- See Cormen et al. for a proof: Its several pages; statistics necessary; some "magical" estimations used.

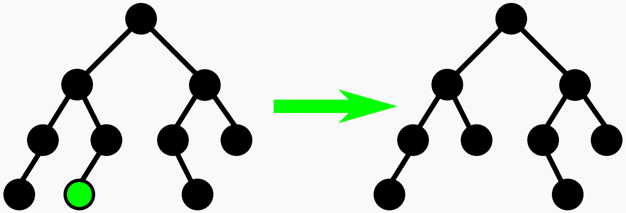
Delete

- Search the node that contains the key we want to delete.
- If no such node exists: do nothing.
- Otherwise we have three cases:
  1. Deleting a Leaf.
  2. Deleting an inner node (but not the root).
  3. Deleting the root.
- Problem: Resulting tree still must obey the binary search property.



Delete

Leaf

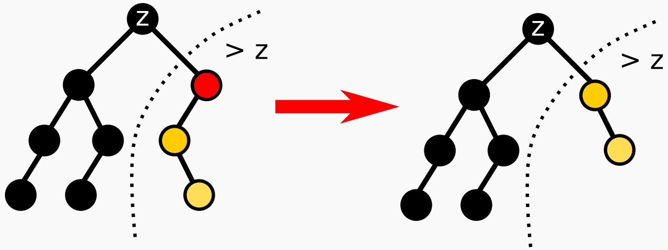


- Remove leaf node: Set reference of parent to null.
- We need the reference to the parent node to do that.
- Binary search tree property still holds.

Delete

Inner node

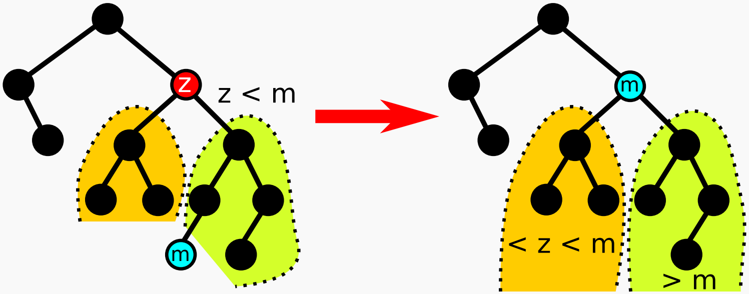
- First case: Inner node only has one sub tree.
- The reference of the parent is replaced by the reference of this sub tree.
- Binary search tree property still holds.



Delete

Inner node

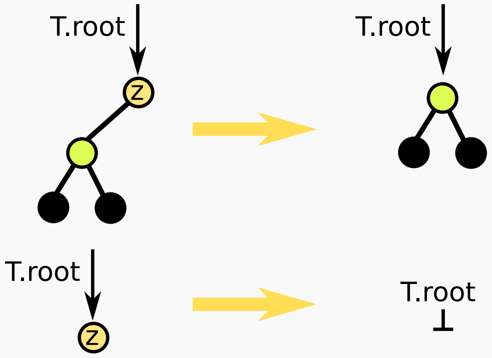
- Second case: Inner node has to sub trees.
- Find the node with the minimum key in the right sub tree.
- Replace the key of the node we want to delete with this minimum.
- Delete the node with the minimum (only first case can occur)
- Binary search tree property still holds.



Delete

Root

- Similiar to the above cases but if the parent reference has to be change, then $T.root$ must be changed.



Delete

Example

Build a binary search tree by inserting the following keys (nodes)
7, 11, 3, 1, 5, 4, 6, 9, 14, 10, 13.
Afterwards remove the keys $1, 6, 5, 11, 7$

Delete

Analysis

No algorithm given (see Cormen).

- Similar to search since the node to be delete has to be found beforehand.
  - Best case: $\Theta(1)$
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(\log n)$

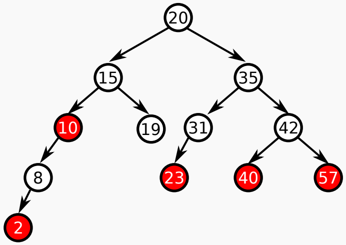
Content

Definition

Definition

Ein **balance red black tree** is a binary search tree such that:

- Every node is either red or black,
- there are no two consecutive red nodes on a path in the tree,
- every path from the root to a leaf contains the same number of black nodes,
- and the root always is black (we still draw its background sometimes white)



Properties

- We use red black trees for short instead of balanced red black trees.
- Search of binary tree can be used for red black trees without modifications.
- In the best case a red black tree contains only black nodes: tree is perfectly balanced. $\Theta(\log_2 n)$ time to search a key in the worst case.
- The worst case is **not** a red black tree with the maximum number of red nodes, since it is still perfectly balanced.
- Worst case: Only one branch contains the maximum number of red nodes. $\Theta(2\log_2 n)$ since the longest branch is two times longer than all other (short) branches.
- Problem: Inserting in $\Theta(\log_2 n)$ possible in all cases? Inserting a node might violate the red black property.
- We restrict the red black property a bit to make implementations easier.

Definition

Left Leaning Red Black Tree

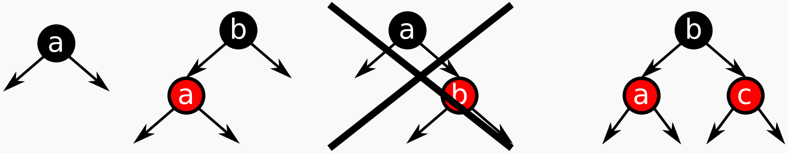
Definition

A **left leaning red black tree** is a red black tree where a right child is only allowed to be red if a red left child exists.

- The previous tree is a left-leaning red black tree.
- Rudolf Bayer, 1972: symmetric binary B-trees. [2]
- Leonidas J. Guibas und Robert Sedgewick, 1976: Rot-Schwarz-Baum.
- Left (right) leaning: Idea by Arne Anderson, 1993. [1]

Implementation

- Unlike Cormen et al: No reference to parent node
- One additional bit (bool) to encode the color of a node x: $x.red = true$ iff the node x is red.



- Hacks to encode color bit: Use the lowest significant bit of the pointer a node, since due to memory alignment it is always zero.
- "swapping pointers": Node is red if the left child contains a key larger then the right child. Search algorithm has to be adapted since the binary search property does not hold anymore.

Insert
Idea

- Recursive implementation, since we do not store the parent node.
- New node is always inserted as a red node.
- Recoloring and restructuring the tree after the recursion while moving "backwards" from leaf to root.
- We need to identify all cases that violate the left leaning red black property.
- We develop the insertion algorithm by step wise refinement of the existing one.
- We assume that x.key is not already contained in the tree.
- Parameter *x* is a red node with empty subtrees.

Listing 22: Insert

```
node = red-black-insert(u, x)
if u = null
  node ← x
else
  if x.key < u.key
    u.left ← red-black-insert(u.left, x)
  else
    u.right ← red-black-insert(u.right, x)
  // treatment of special cases
  node ← u
```

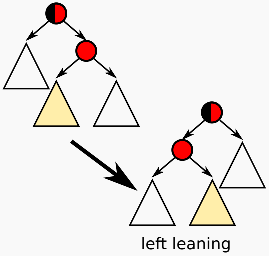
Insert

Second case

- Node is inserted as right child: left leaning property might be violated.
- We use a function red(x) := x != null and x.red

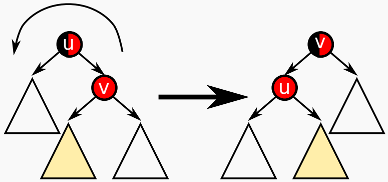
Listing 23: Insert

```
node = red-black-insert(u, x)
if u = null
  node ← x
else
  if x.key < u.key
    u.left ← red-black-insert(u.left, x)
  else
    u.right ← red-black-insert(u.right, x)
  if not red(u.left) and red(u.right)
    u ← red-black-rotate-left(u)
  // treatment of other special cases
  node ← u
```



Insert

Left Rotation



left leaning

```
node = red−black−rotate−left(u)
z = u.right
u.right = z.left
z.left = u
z.red = u.red
u.red = true
node = z
```

- Result of rotation still holds binary search property.
- Number of black nodes on a path unchanged.
- New parent node is returned since its parents left or right reference has to be updated.

Insert

Example

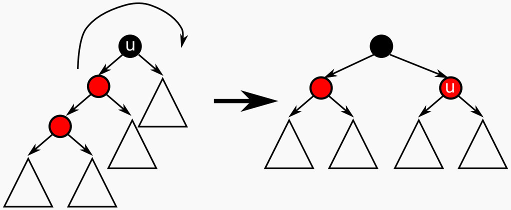
Example

Given a red black tree consisting of a single black root node with key 5.
Insert red nodes 7 and 9.

Insert

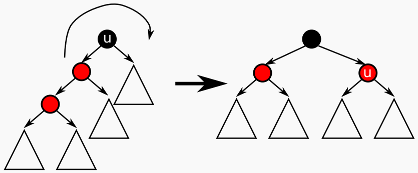
Eliminating two consecutive red nodes

- No two consecutive red nodes can occur on a left path of u due to the prior left rotation or inserting a red node into the left path.
- Solution: Right rotation at the black parent node u.
- Implementation complementary to left rotation.



Insert

Second Case



Listing 24: Insert

```
node = red-black-insert(u, x)
if u = null
  node ← x
else
  if x.key < u.key
    u.left ← red-black-insert(u.left, x)
  else
    u.right ← red-black-insert(u.right, x)
  if not red(u.left) and red(u.right)
    u ← red-black-rotate-left(u)
  if red(u.left) and red(u.left.left)
    u ← red-black-rotate-right(u)
  node ← u
```

Example

Example

Given a red black tree consisting of a single black root node with key 1. Insert (red) nodes 2, 3, and 4.

- Red-black property?

Last Case

- Two consecutive red nodes appear on side path in the example.
- We do not traverse this path, therefor no re-balancing is done.
- We do not want to step into another path.
- Solution: If a black node has a red left and right child node, we re-color them. red nodes to black, black node to red.
- Red-black property still holds (same number of black nodes on each path)
- Possible: Parent of new red node could be red, but this is resolved later while traversing up the path.

Insert

Last Case

Listing 25: Insert

```
node = red-black-insert(u, x)
if u = null
  node ← x
else
  if x.key < u.key
    u.left ← red-black-insert(u.left, x)
  else
    u.right ← red-black-insert(u.right, x)
  if not red(u.left) and red(u.right)
    u ← red-black-rotate-left(u)
  if red(u.left) and red(u.left.left)
    u ← red-black-rotate-right(u)
  if red(u.left) and red(u.right)
    u.red, u.left.red, u.right.red ← true, false, false
  node ← u
```

Insert

Final version

- Insertion might change the root.
- We call our function on T.root.
- Afterwards ensure that T.root is black by always overwriting its color (do not forget this in your examples)

Listing 26: Final Insert

```
red-black-insert(T, x)
T.root ← red-black-insert(T.root, x)
T.root.red = false
```

Example

Given an empty red black tree T.
Insert (red) nodes 1, 2, 3, 4, 5, 6, 7, and 8.
Continue to insert 9, 10, etc. if you are faster than your class mates.

Delete

- Similar to binary tree.
- Re-balancing is the same.
- Pure recursive implementation difficult.
- Cormen: Iterative version with parent node. Re-balance function is called from insert and delete function in different cases to avoid redundant code.
- Java java.util.TreeSet: Implementation of a red black tree identically to the version given in Cormen with additional boiler code.

Contents

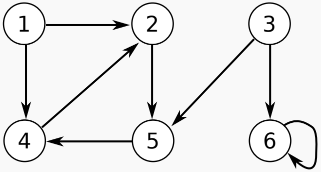
Definition

Definition

A **directed graph, digraph** G is a pair $G = (V, E)$ consisting of a finite set V of **nodes** a set of pairs $E \subseteq V^2$ of **directed edged**.

Example

$G = (V, E)$ with
$V = \{1, 2, 3, 4, 5, 6\}$ und
$E = \{(1, 2), (1, 4), (2, 5), (4, 2),$
$(5, 4), (3, 5), (3, 6), (6, 6)\}$



One possible **drawing** of G.

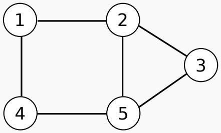
Definition

Definition

An **undirected graph** is a graph $G = (V, E)$, such that for each edge $(u, v) \in E$ also (v, u) is an edge of G.
(v, u) can be in E but does not have to if (u, v) already is contained in E.

Example

$G = (V, E)$ with
$V = \{1, 2, 3, 4, 5\}$ and
$E = \{(1, 2), (1, 4), (2, 3), (2, 5), (3, 5), (4, 5)\}$



A possible **drawing** von G.

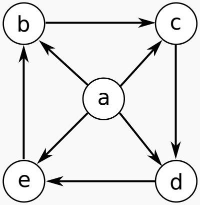
Definition

A **simple graph** only have one edge between two nodes and no edge to itself.

Example

Example

$G = (\{1, 2, 3, 4, 5, 6, 7\}, \{(1, 3), (3, 4), (5, 1), (5, 3), (5, 4), (4, 6), (7, 6), (5, 5)\}$
Draw G as a directed and then as an undirected graph.



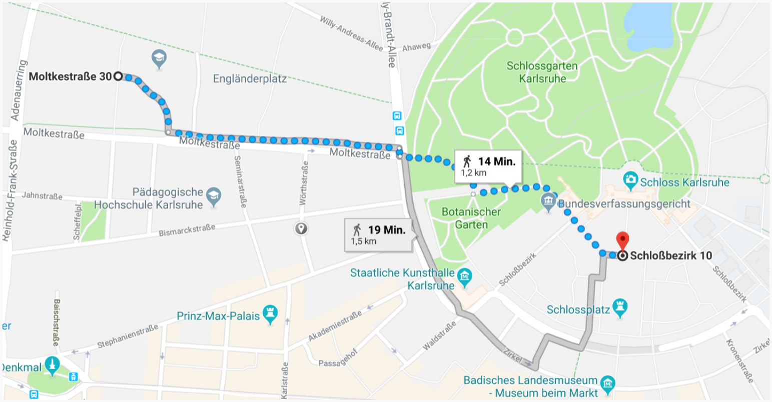
Example

Give the set V and E for the graph $G = (V, E)$ on the left.

Applications

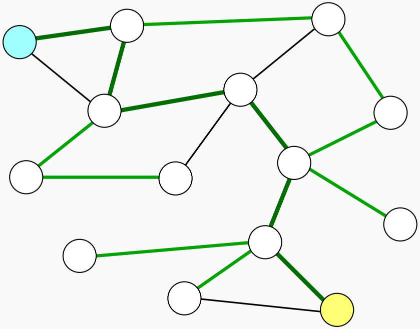
Route Planing

- Find the shortest path, for instance Google Maps
- Algorithm: Single or all shortest path from a given node (your current position) to a destination.



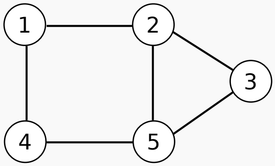
Networks

- Ethernet: only one path between to network nodes is allowed.
- Algorithm: Dynamically finding a minimal spanning tree for a network.
- A spanning tree is a sub graph connecting all nodes but without cycles.
- Minimal: fastest connection, highest capacity, ...



Representation

Undirected graph



- If nodes are given as natural numbers:
  use node number as index in an array.

Adjacency matrix

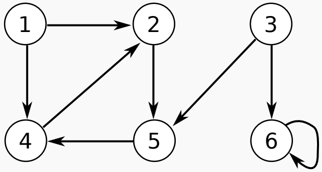
	1	2	3	4	5
1		x		x	
2	x		x		x
3		x			x
4	x				x
5		x	x	x	

Adjacency list

$$\begin{tikzcd}
\boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{4} & \\
\boxed{2} \arrow[r] & \boxed{1} \arrow[r] & \boxed{3} \arrow[r] & \boxed{5} \\
\boxed{3} \arrow[r] & \boxed{5} \arrow[r] & \boxed{2} & \\
\boxed{4} \arrow[r] & \boxed{1} \arrow[r] & \boxed{5} & \\
\boxed{5} \arrow[r] & \boxed{4} \arrow[r] & \boxed{3} \arrow[r] & \boxed{2}
\end{tikzcd}$$

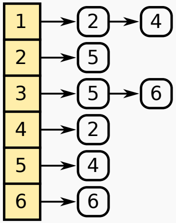
Representation

Digraph



- If nodes are not natural numbers: Create a map from nodes to natural numbers. For instance with hash tables or any other dictionary data structure.

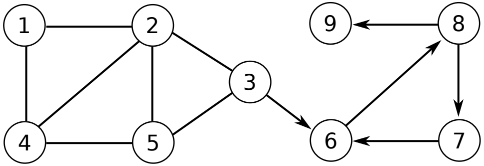
	1	2	3	4	5	6
1		x		x		
2					x	
3					x	x
4		x				
5				x		
6						x



Representation

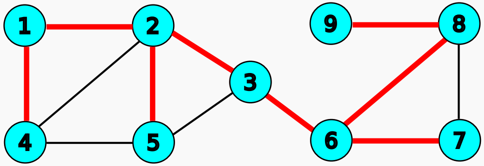
Example

Give an adjacency matrix and an adjacency list for the following graph.

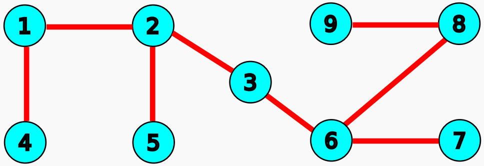


- Represent undirected edges with both directions.

Spanning Tree



- A **spanning tree** is a sub tree $G' = (V, E')$ of a given (undirected, simple) graph $G = (V, E)$ with $E' \subseteq E$, such that G' does not contain a cycle.
- We only consider **connected** graphs: there always is a path from each node to another.



Weighted Graphs

- We add a weight to each edge. it could represent the distance of an edge on a map or the capacity of a network line.
- **(edge) weighted graph** (undirected and connected in our case)
- The weight can be either given as a function $w(e)$ for each edge or as an attribute $e.weight$.
- In drawings of a graph the weight is written near the edge.

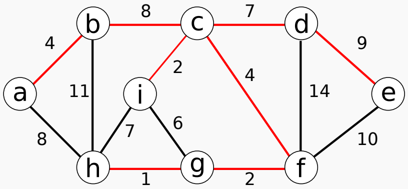
]

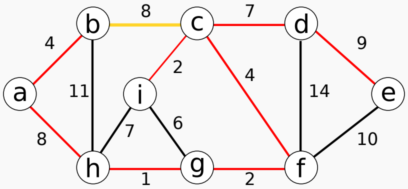
Content

Minimal Spanning Tree

Definition

Minimal spanning tree (MST): A spanning tree of a graph G where the sum of all edge weights is minimal over all spanning trees of G.

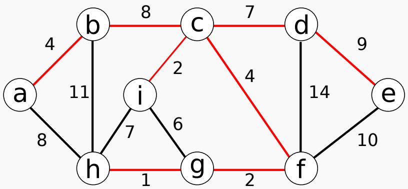


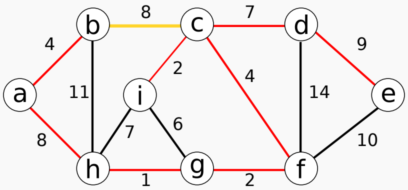


Algorithm

Definition

Minimal spanning tree (MST): A spanning tree of a graph G where the sum of all edge weights is minimal over all spanning trees of G.



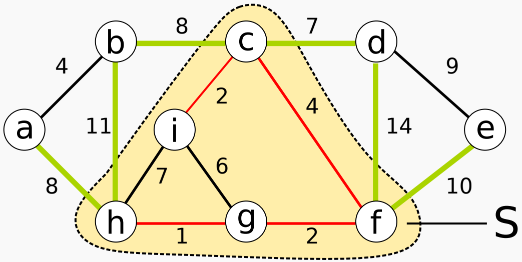


Cut

Definition

Let $G = (E, V)$be an undirected graph and $S \subseteq V$.
The partitioning $(S, V - S)$ is called **cut** trough G.
Each edge $(u, v) \in E$ with $u \in S$ and $v \in V - S$ is called **crossing edge**.

The crossing edges connect the sub graphs formed by S and $V - S$.



Prim's Algorithm

Idea

- In the previous example the crossing edge with minimal weight 7 is part of a MST.
- Starts from an initial cut consisting of a single node (arbitrary).
- Search a crossing edge e with minimal weight among all crossing edges.
- Extend the cut by e.
- Repeat until the result is a spanning tree.
- Theorem: Result is minimum spanning tree (without proof).

Prim's Algorithm

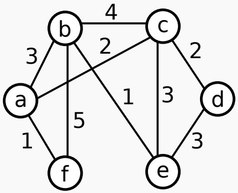
Arbitrary start node r.

Listing 27: Sketch of Prim's Algorithm

```
A = prims-algorithm(V, E, r)
A ← ∅
$V_A$ ← $\{r\}$
while $A$ is not a spanning tree
 find a crossing edge (u,v) with minimal weight in $(V_A, V - V_A)$
  A ← A ∪ { (u,v) }
  $V_A$ ← $V_A$ ∪ { v }
```

Example

Start with a
Give A und V_A after each iteration of the while loop.

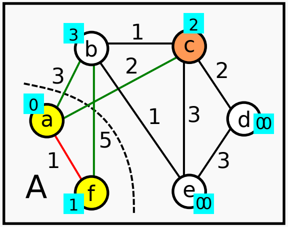


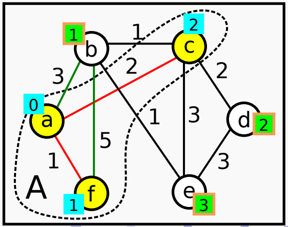
Prim's Algorithm

Implementation

Searching the minimal crossing edge by enumerating every edge is too time consuming. We use a priority queue for all nodes in V_A.

- Each node is added to the queue with initial weight ∞. The starting nodes weight is 0.
- The weight of a node is the weight of its currently smallest crossing edge.
- The currently smallest crossing edge for each node n is stored in an attribute $n.crossing$.
- If the minimal node n is removed from the queue, then $n.crossing$ contains a minimal crossing edge.
- All edges (n, v) are enumerated for this node. If it has a smaller weight than $v.crossing$, then the attributes of v are updated.

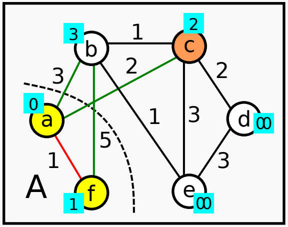


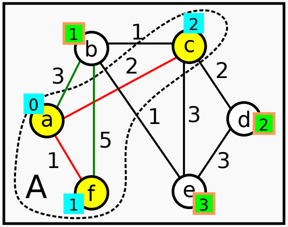


Prim's Algorithm

Listing 28: Prim's Algorithm

```
A = prims-algorithm(E, V, r)
A ← ∅
q ← empty priority queue
foreach u ∈ V
  u.weight  ← ∞
  u.crossing ← null
  priority-enqueue(q, u)
priority-decrease-value(q, r, 0)
while not empty(q)
  u ← priority-dequeue(q)
  if u.crossing ≠ null
    A ← A ∪ { u.crossing }
  foreach (u,v) ∈ E
    if (u,v).weight < v.weight
      decrease(q, v, (u,v).weight)
      v.crossing ← (u,v)
```





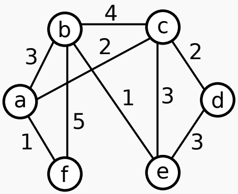
Prim's Algorithm

Worst Case Analysis

- Creating the priority queue with a binary heap bottom up is done in linear Time $\Theta(|V|)$
- Each node is only removed once: $\Theta(|V| \log_2 |V|)$ with a binary heap.
- Each edge is only enumerated once (for each direction): $\Theta(|E|)$ with an adjacency list implementation.
- With a binary heap reducing the weight cost logarithmic for each edge is the *dominating* operation: $\Theta(|E + |E| \log_2 |E|) = \Theta(|V|^2 \log_2 |V|)$
- With a Fibonacci heap: $\Theta(|E| + |V| \log |V|)$ since reducing the weight can be done in constant time (amortized). This is an optimal implementation.
- In practice: Binary heaps are used since implementations of Fibonacci heaps have a significant larger constant factor.

Prim's Algorithm

Example



Example

Start with node *a*.

You do not have to build a binary heap. Either

- draw the graph with node weights, mark the current smallest crossing edge and mark nodes removed from the queue
- or use a table for the current node weight and another one for the smallest crossing edge. Mark nodes that are removed from the queue.

Contents

Arne Andersson.
Balanced search trees made simple.
In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures*, pages 60–71, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

Rudolf Bayer.
Symmetric binary b-trees: Data structure and maintenance algorithms.
Acta Informatica, 1(4):290–306, Dec 1972.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms.
MIT Press, 3. edition, 2009.

Edsger Wybe Dijkstra.
A Discipline of Programming.
Prentice Hall, Inc., 1976.

Rudolf Haller, editor.
Euklid: Elemente – die Stoicheia.
Edition Opera-Platonis, Markgrönigen, 2010.

Charles Anthony Richard Hoare.
Algorithm 64: Quicksort.
Commun. ACM, 4(7):321, 1961.