9.3 Choosing an algorithm

The key factor in choosing an algorithm is usually its **efficiency in that situation**. However, there are other factors:

- implementation is easy
  - is there a suitable algorithm already available?
  - is the advantage of improved efficiency worth the effort of implementing a more complex algorithm?
  - simpler code may not contain errors as easily
  - a simpler solution is easier to maintain
- precision of the results
  - with real numbers round-off errors can be a significant problem
- variation in the running-time
  - e.g. in signal processing the running-time must not vary at all

The programming environment also sets its limitations:

- many languages require that a maximum size is defined for arrays

  $\Rightarrow$ algorithms using arrays get a compile time, artificial upper limit

  - with list structures and dynamic arrays the algorithm works as long as there is memory available in the computer

- the memory can suddenly run out with lists, but not with arrays of a fixed size

  $\Rightarrow$ list structures are not always suitable for embedded systems

- in some computers the space reserved for recursion is much smaller than the space for the rest of the data

  $\Rightarrow$ if a lot of memory is needed, a non-recursive algorithm (or implementation) must be chosen

If the efficiency is the primary factor in the selection, at least the following should be taken into account:

- Is the size of the input data so large that the asymptotic efficiency gives the right idea about the overall efficiency?
- Can the worst-case be slow if the efficiency is good in the average case?
- Is memory use a factor?
- Is there a certain regularity in the input that can be advantageous?
  - with one execution?
  - with several executions?
- Is there a certain regularity in the input that often is the worst-case for some algorithm?

Example: contacts

- operations
  - finding a phonenumber based on the name
  - adding a new contact into the phone-book
  - removing a contact and all the related information
- assumptions
  - additions and removals are needed rarely
  - phonenumber queries are done often
  - additions and removals are done in groups

1st attempt: unsorted array

Virtanen	Järvinen		Lahtinen	
123 555	123 111		123 444	
1	2		n	

- Adding a new name to the end: $O(1)$.
- Searching by scanning the elements from the beginnning (or end): $O(n)$.
- Removing by moving the last element to the place of the removed element: $O(1)$ + search costs = $O(n)$.

$\Rightarrow$ The solution is not suitable since the operations that are needed often are slow.

2nd attempt: sorted array, 1st version

- Adding the new names directly to their correct location in the alphabetical order. The rest of the elements are moved one step forwards: $O(n)$.
- Removing by moving the elements one step backwards: $O(n)$.
- Searching with BIN-SEARCH: $\Theta(\lg n)$.

$\Rightarrow$ The search is efficient but the removal and addition are still slow.

- The solution seems better than the first attempt if our original assumption is correct and searches are done more frequently than additions and removals.

3rd attempt: an almost sorted array

- Keep most of the array sorted and a small unsorted segment at the end of the array (size $O(1)$).
- Additions are done to the segment at the end: $O(1)$.
- Search is done first with binary search in the sorted part and then if needed by scanning the unsorted part at the end: $O(\text{lg } n)$ + $O(l)$.
- Removal is done by leaving the name in and changing the number to 0: $O(1)$ + search costs = $O(\text{lg } n)$ + $O(l)$.
- When the unsorted end segment has become too large, sort the entire array: $\Theta(n \lg n)$.
- A mediocre solution, but
  - $\Theta(l)$ can be large
  - sorting every now and then costs time

4th attempt: a sorted array, 2nd. version

- Let's use the information that additions and removal are done in groups to our advantage.
- Sort the groups of additions and removals.
- Merge the array and the group of additions (like with MERGE) by removing the elements in the group of removals simultaneously.
- Now the search is still logarithmic.
- Addition and removal uses $O(l \lg l)$ + $O(p \lg p)$ + $O(n)$, when l is the amount of additions and p is the amount of removals
- Pretty good!

The problem could also be solved with dynamic sets covered later on the course and naturally, with programming languages' library implementations such as the C++ containers.

9.4 Hash table

The basic idea behind hash tables is to reduce the range of possible key values in a dynamic set by using a *hash function* h so that the keys can be stored in an array.

- the advantatage of an array is the efficient, constant-time indexing it provides

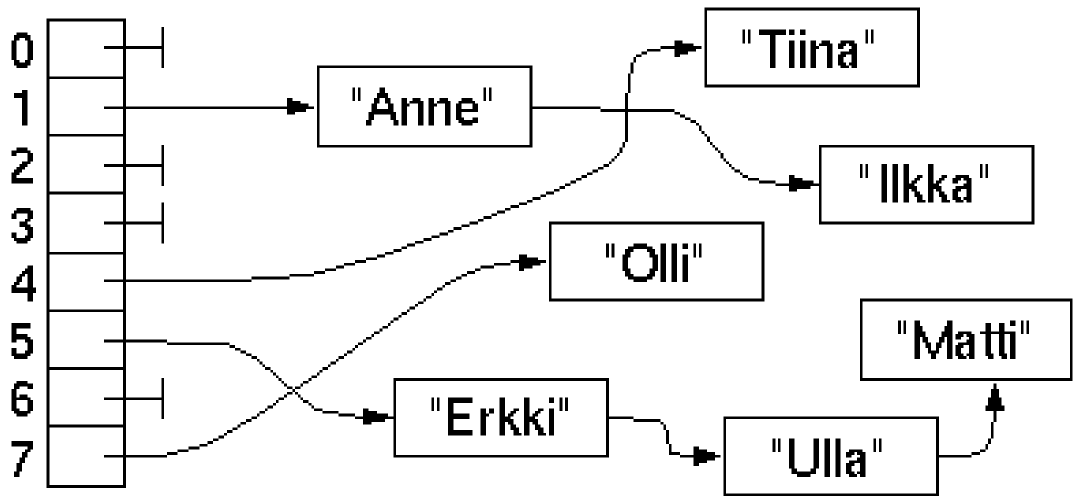
Reducing the range of the keys creates a problem: *collisions*.

- more than one element can hash into the same slot in the hash table

The most common way to solve the problem is called *chaining*.

- all the elements that hash to the same slot are put into a linked list
- there are other alternatives
  - in *open addressing* the element is put into a secondary slot if the primary slot is unavailable
  - in some situations the range of key values is so small, that it doesn't need to be reduced and therefore there are no collisions either
  - this *direct-access table* is very simple and efficient
  - this course covers hashing with chaining only

The picture below shows a chained hash table, whose keys have been hashed based on the first letter according the the table given.



$h(k)$	first letter			
0		H	P	X
1	A	I	Q	Y
2	B	J	R	Z
3	C	K	S	Ä
4	D	L	T	Ö
5	E	M	U	Å
6	F	N	V	
7	G	O	W	

Is this a good hash?

- No. Let's see why.

The chained hash table provides the *dictionary* operations only, but those are very simple:

```
CHAINED-HASH-SEARCH($T, k$)
    ▷ find the element with key $k$ from the list $T[h(k)]$

CHAINED-HASH-INSERT($T, x$)
    ▷ add $x$ to the beginning of the list $T[h(x{\rightarrow}key)]$

CHAINED-HASH-DELETE($T, x$)
    ▷ remove $x$ from the list $T[h(x{\rightarrow}key)]$
```

Running-times:

- addition: $\Theta(1)$
- search: worst-case $\Theta(n)$
- removal: if the list is doubly-linked $\Theta(1)$; with a singly linked list worst-case $\Theta(n)$, since the predecessor of the element under removal needs to be searched from the list
  - in practise the difference is not significant since usually the element to be removed needs to be searched from the list anyway

The *average* running-times of the operations of a chained hash table depend on the lengths of the lists.

- in the worst-case all elements end up in the same list and the running-times are $\Theta(n)$
- to determine the average-case running time we´ll use the following:
  - m = size of the hash table
  - n = amount of elements in the table
  - α = $\frac{n}{m}$ = *load factor* i.e. the average length of the list
- in addition, in order to evaluate the average-case efficiency an estimate on how well the hash function h hashes the elements is needed
  - if for example $h(k)$ = the 3 highest bits in the name, all elements hash into the same list
  - it is often assumed that all elements are equally likely to hash into any of the slots
  - *simple uniform hashing*
  - we´ll also assume that evaluating $h(k)$ is $\Theta(1)$

- if an element that is not in the table is searched for, the entire list needs to be scanned through

  $\Rightarrow$ on average α elements need to be investigated

  $\Rightarrow$ the running-time is on average $\Theta(1+\alpha)$

- if we assume that any of the elements in the list is the key with the same likelyhood, on average half of the list needs to be searched through in the case where the key is found in the list

  $\Rightarrow$ the running-time is $\Theta(1+\frac{\alpha}{2})$ = $\Theta(1+\alpha)$ on average

- if the load factor is kept under some fixed constant (e.g. α < 50 %), then $\Theta(1+\alpha) = \Theta(1)$

  $\Rightarrow$ all operations of a chained hash table can be implemented in $\Theta(1)$ running-time on average

  – this requires that the size of the hash table is around the same as the amount of elements stored in the table

When evaluating the average-case running-time we assumed that the hash-function hashes evenly. However, it is in no way obvious that this actually happens.

The quality of the hash function is the most critical factor in the efficiency of the hash table.

Properties of a good hash function:

- the hash function must be deterministic
  - otherwise an element once placed into the hash table may never be found!
- despite this, it would be good that the hash function is as "random" as possible
  - $\frac{1}{m}$ of the keys should be hashed into each slot as closely as possible

- unfortunately implementing a completely evenly hashing hash function is most of the time impossible
  - the probability distribution of the keys is not usually known
  - the data is usually not evenly distributed
    - ∗ almost any sensible hash function hashes an evenly distributed data perfectly
- Often the hash function is created so that it is independent of any patterns occuring in the input data, i.e. such patterns are broken by the function
  - for example, single letters are not investigated when hashing names but all the bits in the name are taken into account

- two methods for creating hash functions that usually behave well are introduced here
- lets assume that the keys are natural numbers 0, 1, 2, . . .
  - if this is not the case the key can be interpreted as a natural number
  - e.g. a name can be converted into a number by calculating the ASCII-values of the letters and adding them together with appropriate weights

Creating hash functions with the *division method* is simple and fast.

- $h(k) = k \bmod m$
- it should only be used if the value of m is suitable
- e.g. if $m = 2^b$ for some $b \in N = \{0, 1, 2, \ldots\}$, then

$$h(k) = k\text{'s } b \text{ lowest bits}$$

$\Rightarrow$ the function doesn't even take a look at all the bits in k
$\Rightarrow$ the function probably hashes binary keys poorly

- for the same reason, values of m in the format $m = 10^b$ should be avoided with decimal keys
- if the keys have been formed by interpreting a character string as a value in the 128-system, then m = 127 is a poor choise, as then all the permutations of the same string end up into the same slot
- prime numbers are usually good choises for m, provided they are not close to a power of two
  - e.g. $\approx$ 700 lists is needed $\Rightarrow$ 701 is OK
- it's worth checking with a small "real" input data set whether the function hashes efficiently

The *multiplication method* for creating hash functions doesn't have large requirements for the values of m.

- the constant A is chosen so that $0 < A < 1$
- $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$
- if $m = 2^b$, the word length of the machine is w, and k and $2^w \cdot A$ fit into a single word, then $h(k)$ can be calculated easily as follows:

$$h(k) = \lfloor \frac{(((2^w \cdot A) \cdot k) \bmod 2^w)}{2^{w-b}} \rfloor$$

- which value should be chosen for A?
  - all of the values of A work at least somehow
  - the rumor has it that $A \approx \frac{\sqrt{5}-1}{2}$ often works quite well

10 Trees

http://imgur.com/L77FY5X

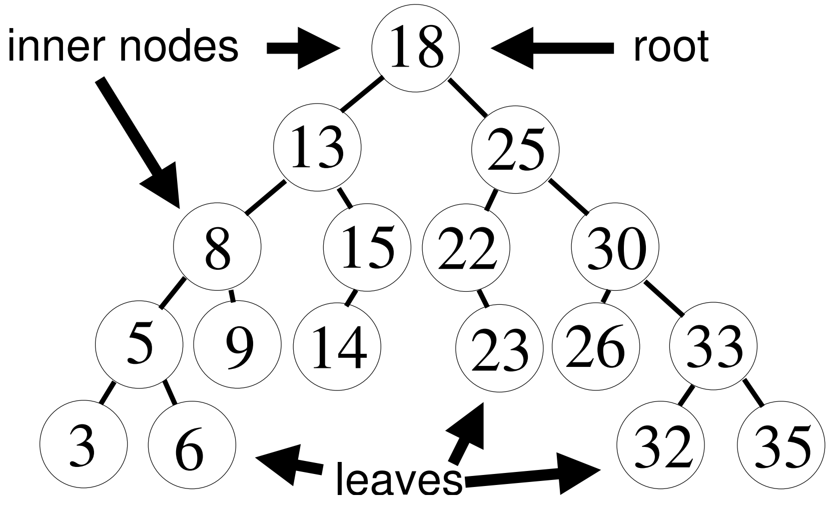
This chapter discusses the most commonly used tree structures.

- first an ordinary binary search tree is covered and then it is balanced by converting it into a red-black tree.
- the B-tree is introduced as an example of a tree whose nodes can have more than two children
- Splay trees and AVL-trees are mentioned

10.1 Basic binary search tree

A *binary tree* is a finite structure of *node*s that is either

- empty, or
- contains a node called the *root*, and two binary trees called the *left subtree* and the *right subtree*
- childless nodes are called *leaves*
- other nodes are *internal nodes*
- a node is the *parent* of its children
- the *ancestor*s of a node are the node itself, its parent, the parents parent etc.
- a *descendant* is defined similarly



Additionally, all elements in a *binary* ***search*** *tree* satisfy the following property:

Let l be any node in the left subtree of the node x and r be any node in the right subtree of x then

$$l.key \leq x.key \leq r.key$$

- the binary tree in the picture on the previous page is a binary search tree
- the heap described in chapter 3.1 is a binary tree but not a binary search tree

Usually a binary tree is represented by a linked structure where each object contains the fields key, $left$, $right$ and p (parent).

- there may naturally be satellite data

Searching in a binary search tree

- search in the entire tree R-TREE-SEARCH($T.root, k$)
- returns a pointer x to the node with $x{\rightarrow}key = k$, or NIL if there is no such node

```
R-TREE-SEARCH($x, k$)
1 if $x$ = NIL or $k = x{\rightarrow}key$ then
2     return $x$                                   (the searched key is found)
3 if $k < x{\rightarrow}key$ then                    (if the searched key is smaller...)
4     return R-TREE-SEARCH($x{\rightarrow}left, k$)    (...search from the left subtree)
5 else                                             (else...)
6     return R-TREE-SEARCH($x{\rightarrow}right, k$)   (...search from the right subtree)
```

The algorithm traces a path from the root downward.

In the worst-case the path is down to a leaf at the end of the longest possible path.

- running-time $O(h)$, where h is the height of the tree
- extra memory requirement $O(h)$ due to recursion

The search can be done without the recursion (which is recommendable).

- now the extra memory requirement is $\Theta(1)$
- the running-time is still $O(h)$

```
TREE-SEARCH($x, k$)
1  while $x \neq$ NIL and $k \neq x\rightarrow key$ do    (until the key is found or a leaf is reached)
2      if $k$ < $x\rightarrow key$ then                    (if the searched key is smaller...)
3          $x$ := $x\rightarrow left$                      (...move left)
4      else                                              (else...)
5          $x$ := $x\rightarrow right$                     (...move right)
6  return $x$                                            (return the result)
```

Minimum and *maximum*:

- the minimum is found by stepping as far to the left as possible

```
TREE-MINIMUM($x$)
1  while $x \rightarrow left \neq$ NIL do
2      $x$ := $x \rightarrow left$
3  return $x$
```

- the maximum is found similarly by stepping as far to the right as possible

```
TREE-MAXIMUM($x$)
1  while $x \rightarrow right \neq$ NIL do
2      $x$ := $x \rightarrow right$
3  return $x$
```

- the running-time is $O(h)$ in both cases and extra memory requirement is $\Theta(1)$

The structure of the tree can be used to our advantage in finding the *successor* and the *predecessor* of a node

- this way SCAN-ALL works correctly even if the elements in the tree all have the same key $\Rightarrow$ an algorithm that finds the node which is the next largest node in inorder from the node given
- one can be build with the TREE-MINIMUM algorithm
- the successor of the node is either
  - the smallest element in the right subtree
  - or the first element on the path to the root whose left subtree contains the given node
- if such nodes cannot be found, the successor is the last node in the tree

```
TREE-SUCCESSOR($x$)
1   if $x \rightarrow right \neq$ NIL then              (if there is a right subtree...)
2       return TREE-MINIMUM($x \rightarrow right$)      (...find its minimum)
3   $y$ := $x \rightarrow p$                            (else step towards the root)
4   while $y \neq$ NIL and $x$ = $y \rightarrow right$ do   (until we´ve moved out of the left child)
5       $x$ := $y$
6       $y$ := $y \rightarrow p$
7   return $y$                                          (return the found node)
```

- note, the keys of the nodes are not checked!
- cmp. finding the successor from a sorted list
- running-time $O(h)$, extra memory requirement $\Theta(1)$
- TREE-PREDECESSOR can be implemented similarly

TREE-SUCCESSOR and TREE-MINIMUM can be used to scan the tree in inorder

```
TREE-SCAN-ALL($T$)
1  if $T.root \neq$ NIL then
2      $x$ := TREE-MINIMUM($T.root$)        (start from the tree minimum)
3  else
4      $x$ := NIL
5  while $x \neq$ NIL do                   (scan as long as there are successors)
6      process the element $x$
7      $x$ := TREE-SUCCESSOR($x$)
```

- each edge is travelled through twice, into both directions
  $\Rightarrow$ TREE-SCAN-ALL uses only $\Theta(n)$ time, although it calls TREE-SUCCESSOR n times

- extra memory requirement $\Theta(1)$

  $\Rightarrow$ TREE-SCAN-ALL is asymptotically as fast as and has a better memory consumption than INORDER-TREE-WALK

  - the difference in constant coefficients is not significant

  $\Rightarrow$ it's worth choosing TREE-SCAN-ALL, if the nodes contain p-fields
- TREE-SCAN-ALL allows several simultaneous scans while INORDER-TREE-WALK doesn't

Insertion into a binary search tree:

```
TREE-INSERT($T, z$)                          ($z$ points to a structure allocated by the user)
y := NIL; x := T.root                        (start from the root)
**while** $x \neq$ NIL **do**                (go downwards until an empty spot is located)
    $y := x$                                 (save the potential parent)
    **if** $z \rightarrow key < x \rightarrow key$ **then**   (move left or right)
        $x := x \rightarrow left$
    **else**
        $x := x \rightarrow right$
$z \rightarrow p := y$                       (make the node found the parent of the new one)
**if** $y$ = NIL **then**
    $T.root := z$                            (the root is the only node in the tree)
**else if** $z \rightarrow key < y \rightarrow key$ **then**  (make the new node its parent's left ...)
    $y \rightarrow left := z$
**else**                                     (... or right child)
    $y \rightarrow right := z$
$z \rightarrow left$ := NIL; $z \rightarrow right$ := NIL
```

The algorithm traces a path from the root down to a leaf, a new node is always placed as a leaf.
$\Rightarrow$ running-time $O(h)$, extra memory requirement $\Theta(1)$

Deletion is more compilicated as an internal node may also be removed:

```
TREE-DELETE($T, z$)                                      (*$z$ points to the node under deletion*)
if $z \rightarrow left = $ NIL or $z \rightarrow right = $ NIL then   (*if $z$ has only one child . . .*)
    $y$ := $z$                                           (*. . . make $z$ the stucture to be removed*)
else
    $y$ := TREE-SUCCESSOR($z$)                           (*else remove the successor of $z$*)
if $y \rightarrow left \neq$ NIL then                    (*save the only child*)
    $x$ := $y \rightarrow left$
else
    $x$ := $y \rightarrow right$
if $x \neq$ NIL then                                     (*if the child exists . . .*)
    $x \rightarrow p$ := $y \rightarrow p$               (*. . . link it into the place of the removed node*)
if $y \rightarrow p$ = NIL then                          (*if the removed node was the root . . .*)
    $T.root$ := $x$                                      (*. . . make $x$ the new root*)
else if $y = y \rightarrow p \rightarrow left$ then      (*set $x$ into the place of the removed node . . .*)
    $y \rightarrow p \rightarrow left$ := $x$            (*. . . as the left child . . .*)
else
    $y \rightarrow p \rightarrow right$ := $x$           (*. . . or the right child or its parent*)
if $y \neq z$ then                                       (*if a node other than z was removed $z$ . . .*)
    $z \rightarrow key$ := $y \rightarrow key$           (*. . . switch the data of $z$ and the removed noc
    $z \rightarrow satellitedata$ := $y \rightarrow satellitedata$
return $y$                                               (*return a pointer to the removed node*)
```

Note! It is really known on line 5 that y only has one child.

- if z has only one child y is z
- if TREE-SUCCESSOR is called on line 4, it is known that z has a right subtree, whose minimum is y
  - minimum cannot have a left child

The algorithms seems complex but all operations besides TREE-SUCCESSOR on line 4 are constant time.
$\Rightarrow$ running-time is $O(h)$ and the extra memory requirement Θ(1)

All basic operations of a dynamic set can be implemented with a binary search tree in $O(h)$ time and with $\Theta(1)$ extra memory:

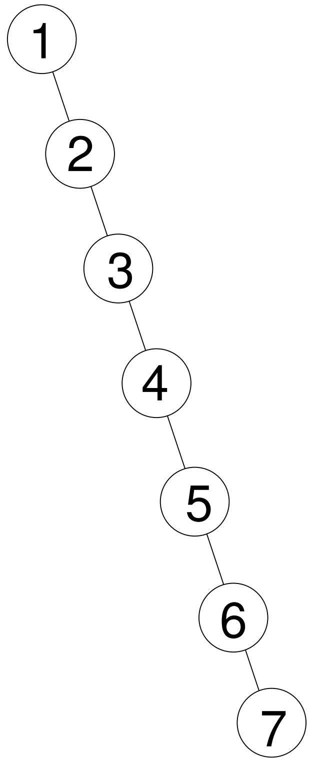
SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR

How high are binary search trees usually?

If we assume that the elements have been entered in a random order and every order is equally probable, the height of a binary search tree build solely with INSERT is on average $\Theta(\lg n)$.
$\Rightarrow$ all operations are on average $\Theta(\lg n)$

Unfortunately the result is not neary as efficient if the keys are entered in an ascending order, as you can see in the picture.

- the height is n -1, lousy!



The problem cannot be solved sensibly with randomization, if all the operations of a dynamic set need to be maintained.

The solution is keeping the tree balanced, we'll return to that later.

Binary search trees - like other data structures - can be made suitable for new tasks by adding new fields essential to the new problem into the structures.

- the basic operations also need to be updated to maintain the contents of the new fields.
- for example, adding a field to the nodes that tells the height of the subtree
  - a function that returns the element i in linear time relative to the height of the tree can be implemented
  - a function that in linear time relative to the height of the tree tells the sequence number of the element in the ascending order
  - the algorithms would be much more inefficient, linear to the amount of the nodes, without the extra fields