Data Structures

Bengt J. Nilsson

Malmö University, Sweden

Today's Content

Introduction to Data Structures

"A *data type* defines the kind of value a variable can hold, such as integers or strings, while a *data structure* is a way to organize and store multiple data types together, like arrays or linked lists. Essentially, data types are the building blocks, and data structures are the frameworks that hold those blocks together."

A data structure is collection of data elements that are organized and/or stored in a way that makes certain operations easier/faster and more efficient.

In some programming languages such constructions are already implemented, in others they need to be built using simpler constructions and the *primitive data types*.

primitive data types:
integer, **float/real**, **boolean**, **character**.

Composite Data Types: Records

A way to collect data elements that together make up parameters for something that can be viewed as a unit. In C these are called *structs*.

Type definition:

type record
  other-type-1: *name-1*
  other-type-2: *name-2*
    ⋮
  other-type-n: *name-n*
end record type-name

Type use:

var type-name *variable*
  ⋮
variable.*name-2* $\leftarrow$ "Nilsson, B.J.";

Access costs $O(1)$ time!

Composite Data Types: Records

A way to collect data elements that together make up parameters for something that can be viewed as a unit. In C these are called *structs*.

Type definition:

type record
    other-type-1: *name-1*
    other-type-2: *name-2*
        ⋮
    other-type-n: *name-n*
end record type-name

Type use:

var type-name *variable*
    ⋮
variable.*name-2* ← "Nilsson, B.J.";

Access costs $O(1)$ time!

We can have nested records:

type record
    integer: *key*
    string: *name*
    record
        string: *addrText*
        string: *postcode*
    end record *address*
end record homeOwner

p is a variable of type homeOwner, access *postcode* by:

p.address.postcode

Composite Data Types: Arrays

A way to organize a sequence of data items.

Type definition:

var type-name[] $V[n]$ ◁ n is an integer

var type-name[][] $W[16][20]$ ◁ 2D array

⋮

var integer[] A ← [4, 3, 8, 14, 0, 0, 6, 0, 9, 0] ◁ explicit array with 10 items

indices	0	1	2	3	4	5	6	7	8	9
values	4	3	8	14	0	0	6	0	9	0

A[4] ← 5 ◁ assigning index 4 position value 5

indices	0	1	2	3	4	5	6	7	8	9
values	4	3	8	14	5	0	6	0	9	0

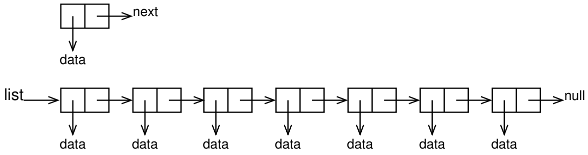
A[12] ← 13 ◁ Error! Index out of range!

W[1][9] ← 'X' ◁ If *type-name* is *char* or *string*

User Defined Data Types: Singly Linked Lists

Type definition:

type record
  dataType: *data*
  sListType: *next*
end record sListType



Algorithm *InsertFirst*
Input: A record r to be inserted first in *list*

r.next $\leftarrow$ *list*
list $\leftarrow$ r

End *InsertFirst*

Takes $O(1)$ time!

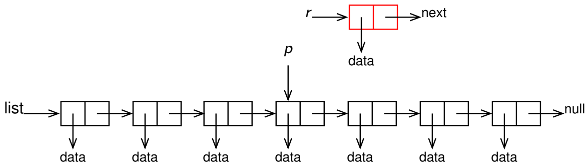
Algorithm *InsertLast*
Input: A record r to be inserted last in *list*

if *list* $=$ null **then**
  list $\leftarrow r$, r.next $\leftarrow$ null
else
  var sListType $p \leftarrow$ *list*
  while p.next $\neq$ null **do** $p \leftarrow p$.next **endwhile**
  p.next $\leftarrow r$, r.next $\leftarrow$ null
endif

End *InsertLast*

Takes $O(n)$ time, if *list* contains n elements!

User Defined Data Types: Singly Linked Lists



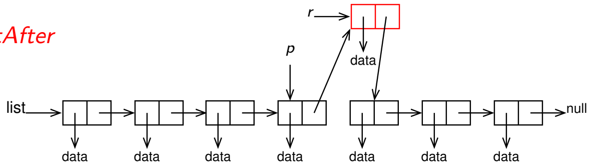
Algorithm *InsertAfter*

Input: A record r to be inserted after p in *list*

r.next $\leftarrow$ p.next

p.next $\leftarrow$ r

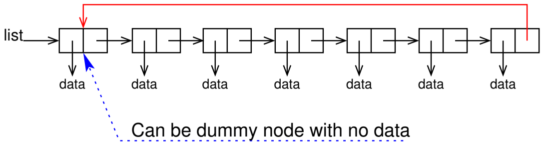
End *InsertAfter*



Takes $O(1)$ time

User Defined Data Types: Circular Singly Linked Lists

To avoid doing null tests, we can make the list circular

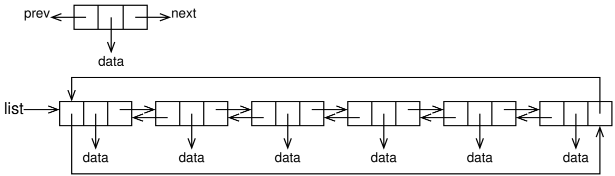


With a dummy node, no need to test for empty list, simplifies the code

User Defined Data Types: Doubly Linked Lists

Type definition:

type record
  dataType: *data*
  dListType: *next*
  dListType: *prev*
end record dListType



Algorithm *InsertAfter*
Input: A record r to be inserted after p in *list*

r.next $\leftarrow$ p.next
p.next.prev $\leftarrow$ r
r.prev $\leftarrow$ p
p.next $\leftarrow$ r

End *InsertAfter*

Takes $O(1)$ time!

Algorithm *InsertBefore*
Input: A record r to be inserted before p in *list*

r.prev $\leftarrow$ p.prev
p.prev.next $\leftarrow$ r
r.next $\leftarrow$ p
p.prev $\leftarrow$ r

End *InsertBefore*

Takes $O(1)$ time!

User Defined Data Types: Linked Lists

Deletion: implemented symmetrically as inserts
(don't forget to deallocate used memory)

$O(1)$ time complexity (except `deleteLast` in SLLs $O(n)$ time)

Search: need to go through list from beginning to end so $O(n)$ time

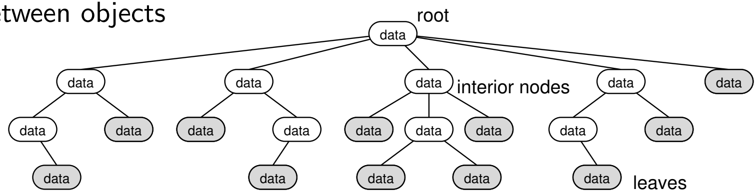
Same as we saw for searching in arrays (previous lecture)

Can we organize data so that searches go faster than $O(n)$ time?

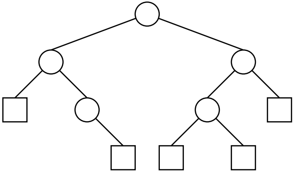
Topic for next time!

User Defined Data Types: Trees

Generic rooted trees: structure that hierarchical relationships between objects



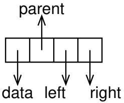
Binary trees: every node has at most two children

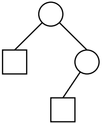


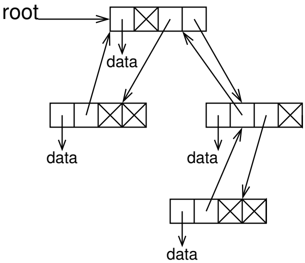
User Defined Data Types: Binary Trees

Type definition:

type record
  dataType: *data*
  bTreeType: *left*
  bTreeType: *right*
  bTreeType: *parent*
end record bTreeType







Insertion: preferably done at leaves (minimizes restructuring), takes $O(1)$ time

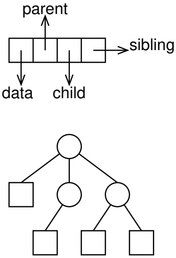
Deletion: requires restructuring the tree, depends on how the tree is organized

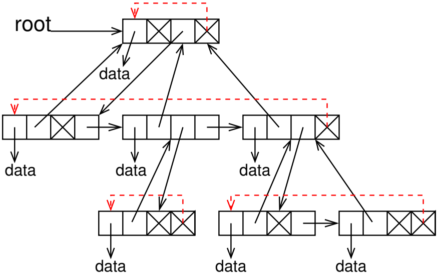
Search: depends on the organization of the tree.
  Worst case — $O(n)$ time, traversal of the tree
  Best case — $O(h)$ time, h is height of the tree

User Defined Data Types: Generic Trees

Type definition:

type record
  dataType: *data*
  gTreeType: *child*
  gTreeType: *sibling*
  gTreeType: *parent*
end record gTreeType





Operations: similar to binary trees

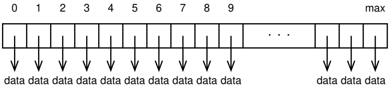
Generic trees allow for varying number of children (constant number)

User Defined Data Types: Trees with Arrays

k-ary trees — node has at most k children

Type definition:

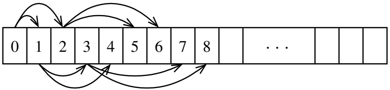
var dataType[] T[max] $\triangleleft$ max is upper bound on n



Root: root is at index 0

Node at index i: children at index $i \cdot k + 1$, $i \cdot k + 2, \ldots, i \cdot k + k$
parent at index $(i - 1) /\!\!/ k$ $\triangleleft$ integer division

Most common for binary trees, $k = 2$



Used to implement the data structure **heap**

Abstract Data Types

“ADT is a logical description and data structure is concrete. ADT is the logical picture of the data and the operations to manipulate the component elements of the data. Data structure is the actual representation of the data during the implementation and the algorithms to manipulate the data elements. ADT is in the logical level and data structure is in the implementation level.”

Abstract Data Types

An ADT is an . . . "Object working on a set of elements, for which the *logical behavior* is defined by a set of operations"

Mathematically: a tuple $(\mathcal{E}, \mathcal{F})$, where $\mathcal{E}$ is a set of elements and $\mathcal{F}$ is a set of operations describing the semantics/behaviour of the ADT

In many cases, the set $\mathcal{E}$ is implicit and not given explicitly: integers, strings, records of personal info,. . .

Later generalized to **Software Design Patterns**: "templates for solving particular types of problems, that can then be deployed in many different situations."
ADTs are common patterns for data organization.

The Sequence or List ADT

Operations: `getFirst`, `getLast`, `getNext`, `getPrevious`, `insertBefore`, `insertAfter`, `delete`
(Constructors & admin: `create`, `isEmpty`, `getSize`)

Keeps a linear sequence of objects:

$$a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n$$

Implemented using:

array (non-dynamic), `inserts` and `deletes` in specific position take $O(n)$ time if element order is important.
Retrievals take $O(1)$ time from specific position

linked list, each operation taking $O(1)$ time in specific position.

We have ignored the search cost! (More on this later!)

The Queue ADT

Operations: `enQ`, `deQ`, (`front`)

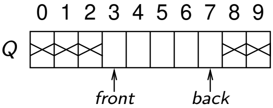
Keeps a FIFO structure

Implemented using array (fixed limit size) or linked list, each operation taking $O(1)$ time

The Queue ADT

Array implementation (problems):

Assume Q is the array, *front* and *back* are the indices



Algorithm *enQ*

Input: enqueue a record r to queue Q

$back \leftarrow back + 1$

$Q[back] \leftarrow r$

End *enQ*

Algorithm *deQ*

Input: returns the record at the front of Q

var dataType $r \leftarrow Q[front]$

$front \leftarrow front + 1$

return r

End *deQ*

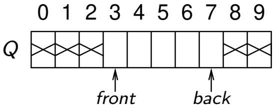
We only increase *front* and *back*! What happens when they reach the end of the array?

Must shift the elements towards the beginning. Expensive, $O(n)$ time!

The Queue ADT

Array implementation (problems):

Assume Q is the array, *front* and *back* are the indices



Algorithm *enQ*

Input: enqueue a record r to queue Q

$back \leftarrow back + 1$

$Q[back] \leftarrow r$

End *enQ*

Algorithm *deQ*

Input: returns the record at the front of Q

var dataType $r \leftarrow Q[front]$

$front \leftarrow front + 1$

return r

End *deQ*

We only increase *front* and *back*! What happens when they reach the end of the array?

Must shift the elements towards the beginning. Expensive, $O(n)$ time!

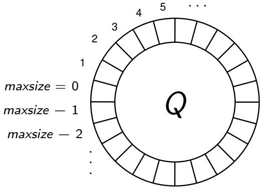
Solution

The Queue ADT

Enter the **Circular Buffer**

Glue end of array Q to the beginning, *maxsize* and 0 are the same index

Maintain *size* and *front* variables



Algorithm *enQ*
Input: enqueue a record r to queue Q

if $size \geq maxsize$ **then** report error **endif**
$size \leftarrow size + 1$
$Q[(front + size) \textbf{ mod } maxsize] \leftarrow r$
End *enQ*

Both take $O(1)$ time

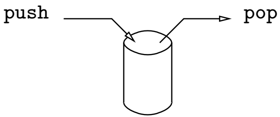
Algorithm *deQ*
Input: returns the record at the front of Q

if $size \leq 0$ **then** report error **endif**
var dataType $r \leftarrow Q[front]$
$front \leftarrow (front + 1) \textbf{ mod } maxsize$
$size \leftarrow size - 1$
return r
End *deQ*

The Stack ADT

Operations: `push`, `pop`, (`top`)

Keeps a LIFO structure



Implemented using array (fixed limit size) or linked list, each operation taking $O(1)$ time

Straightforward, one reference needed to the entry/exit point of the stack

The Stack ADT

We can mitigate the drawback of fixed array size

Type definition:

type record
  dataType[]: $S[256]$
  integer: $size \leftarrow 0$
  integer: $maxsize \leftarrow 256$
end record stackType
var stackType *stack*

$\vdots$

push(*stack*, e_1)
push(*stack*, e_2)

$\vdots$

$e \leftarrow$ pop(*stack*)

Algorithm *push*
Input: *stack* and the record r to push

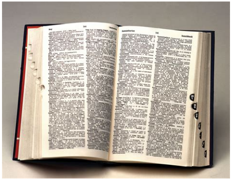
if $stack.size \geq stack.maxsize$ **then**
  $stack.maxsize \leftarrow 2 \cdot stack.maxsize$
  var dataType[] $S[maxsize]$
  for $i \leftarrow 0$ **to** $stack.size - 1$ **do**
    $S[i] \leftarrow stack.S[i]$
  endfor
  $stack.S \leftarrow S$ $\triangleleft$ don't forget to deallocate
endif
$stack.S[size] \leftarrow r$
$stack.size \leftarrow stack.size + 1$
End *push*

Some push operations are expensive, $O(n)$ time, but overall each operation has *amortized cost* $O(1)$

The Dictionary ADT

Operations: `insert`, `delete`, `find`

Keeps a discrete set structure searchable on a *key*



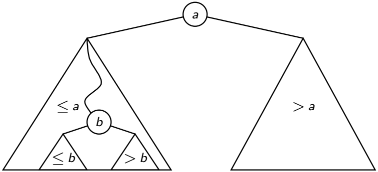
Implementation using array or linked list requires `find` or `insert`/`delete` operations to take $O(n)$ time

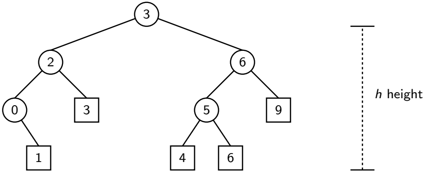
Implementation using balanced binary search tree guarantees $O(\log n)$ time per operation

Implementation using hash table gives $O(1)$ time per operation (probabilistic model)

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key



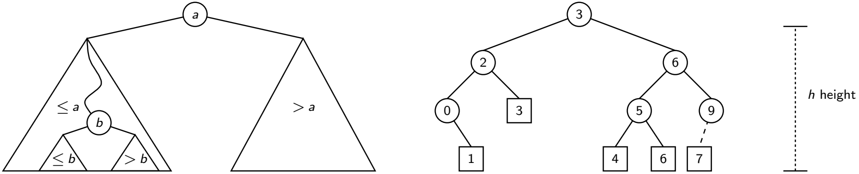


`insert`/`find` compare keys, following left/right children until element found (or insert a new leaf)

Takes $O(h)$ time

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key

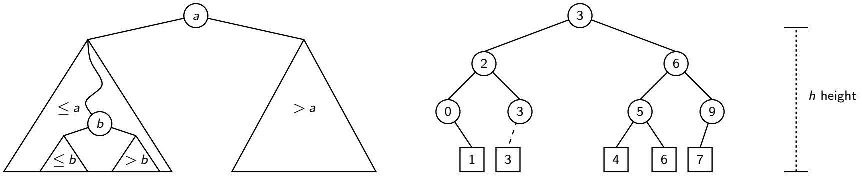


`insert`/`find` compare keys, following left/right children until element found (or insert a new leaf)

Takes $O(h)$ time

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key

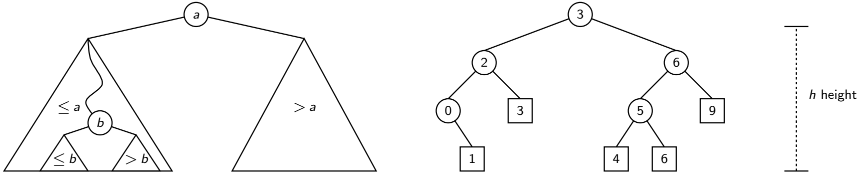


`insert`/`find` compare keys, following left/right children until element found (or insert a new leaf)

Takes $O(h)$ time

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key

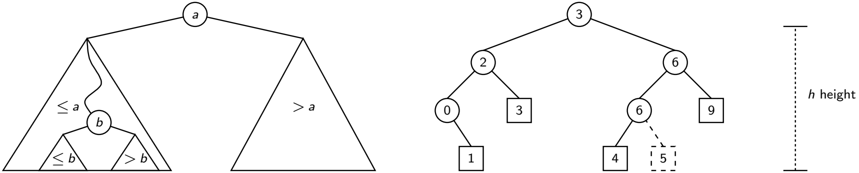


`delete` slightly more complicated
Easy if node has zero or one child! relink child with parent
If node has two children, trick: swap data with leftmost child in right subtree, then remove that node

Takes $O(h)$ time

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key

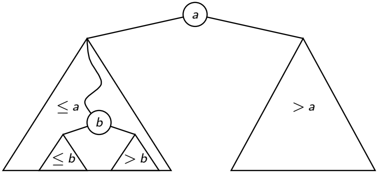


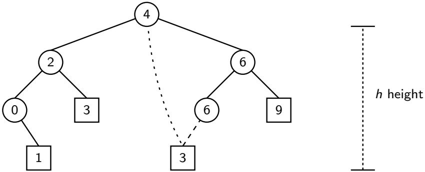
`delete` slightly more complicated
Easy if node has zero or one child! relink child with parent
If node has two children, trick: swap data with leftmost child in right subtree, then remove that node

Takes $O(h)$ time

Dictionaries with Binary Search Trees

Data stored in nodes, ordered so that left subtree has keys $\leq$ node's key, right subtree has keys $>$ node's key





delete slightly more complicated

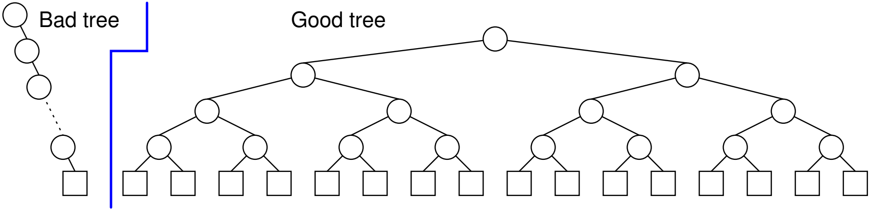
Easy if node has zero or one child! relink child with parent

If node has two children, trick: swap data with leftmost child in right subtree, then remove that node

Takes $O(h)$ time

Dictionaries with Binary Search Trees

How large is h?



Worst case height is $O(n)$ if data inserted in order

Best case height is $O(\log n)$
sum nodes on levels: $1 + 2 + \cdots + 2^h = 2^{h+1} - 1 = n$

Invent balancing schemes: AVL-trees, red-black trees, AA trees,...

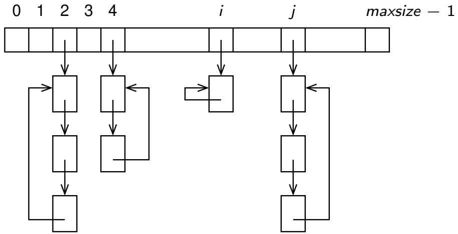
At updates, restructures the tree to maintain enough balance to guarantee $h \in O(\log n)$

Dictionaries with Arrays: Hash Tables

Requires function: $hash(data) \to$ natural number

Type definition:

type record
  cListType[]: $H[1024]$
  integer: $size \leftarrow 0$
  integer: $maxsize \leftarrow 1024$
end record dictType
var dictType *dictionary*



`insert`, `delete`, `find` in circular linked list attached to each position in *dictionary.H*

To access position:

$$dictionary.H[hash(data) \ \mathbf{mod}\ dictionary.maxsize]$$

Dictionaries with Arrays: Hash Tables

Good hash function: should give unique value for each data, over different data behave as random numbers

If $n = \mathit{dictionary.size} < c \cdot \mathit{dictionary.maxsize}$, c constant < 1
$c \approx 0.8$ is popular choice, then

1. Expected size of linked list is $O(1)$
2. Expected size of longest linked list is $O(\log n)$

Variations:

Use doubling scheme as in Stack example when

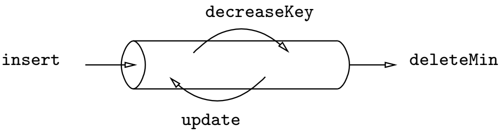
$$\mathit{dictionary.size} \geq c \cdot \mathit{dictionary.maxsize}$$

Use balanced binary search tree instead of circular linked list as secondary structure: longest expected search time $O(\log \log n)$

The Priority Queue ADT

Operations: `insert`, `deleteMin` (`deleteMax`), `update` (`decreaseKey`)

Keeps a FIFO structure on priorities and with update possibility



Implementation using array or linked list requires some operations taking $O(n)$ time

Implementation using binary heap guarantees $O(\log n)$ time per operation

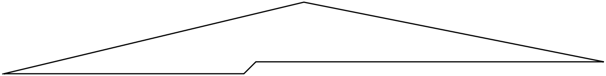
Implementation using Fibonacci heap gives $O(1)$ amortized time for `insert`/`update`

The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in decreasing order

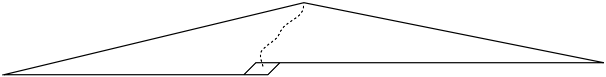


The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in decreasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

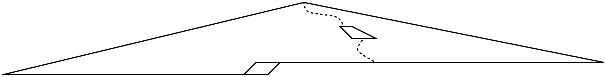
The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.

`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in decreasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

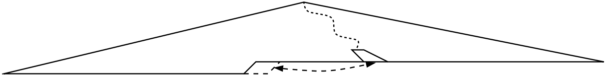
`update`: if key is decreased, do `trickle-up` along parent path. If key is increased, do `trickle-down` along smallest child path

The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in decreasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

`update`: if key is decreased, do `trickle-up` along parent path. If key is increased, do `trickle-down` along smallest child path

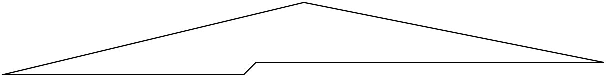
`deleteMin`: place element in last position in root, then do `trickle-down` along smallest child path, return original root

The Priority Queue ADT

Can be implemented using a balanced binary search tree, `insert` and updates as usual.
`deleteMin` returns the leftmost value in the tree.

There is a better way: a **heap**

Perfectly balanced binary tree where path along parents from every leaf to the root is in decreasing order



`insert`: add element in last position of perfectly balanced tree. Do `trickle-up` along parent path

`update`: if key is decreased, do `trickle-up` along parent path. If key is increased, do `trickle-down` along smallest child path

`deleteMin`: place element in last position in root, then do `trickle-down` along smallest child path, return original root

Simplest implementation: array-based binary tree, more next time

Thank you for your attention.

Questions? Comments?