4EE11 – Project Programmeren voor W

College 3, 2008–2009, Blok D

Tom Verhoeff, Software Engineering & Technology, TU/e

Onderwerpen

- Grotere programma's ontwerpen/maken
- Datastructuren en algoritmes

Evolutie, iteratie

- Voeg stapsgewijs functionaliteit toe aan een werkend geheel en zorg er voor dat het daarna nog steeds werkt
- Vooraf bepalen welke uitbreidingen je gaat doen; volgorde kiezen
- Soms moet je iets bestaands eerst wat aanpassen (refactoring)

Ontwerpstijlen

- Bottom-up design: begin met de kleine onderdelen en voeg die stapsgewijs samen tot een groter geheel
- Top-down design: begin met de grote lijn en verfijn die stapsgewijs tot alle onderdelen zijn uitgewerkt

Kanttekeningen

- Bottom-up: wordt makkelijk chaotisch en onoverzichtelijk
- Top-down: in het begin niet veel inzicht in wat goede verfijningen zijn; risico op voorbarige keuzes en op duplicatie van werk
- Beter: yoyo design

Voorbeeld probleem

- Een rij gedateerde meetwaarden (id., time stamp, getal) verwerken: minimum, maximum, gemiddelde en standaarddeviatie bepalen en doorsturen per email naar een centrale

Top-down aanpak

- Topnivo opdeling:
  1. alle meetwaarden inlezen (in globale variabele);
  2. Kengetallen bepalen van meetwaarden
  3. Email boodschap samenstellen met kengetallen en versturen
- Verfijnen: losse meetwaarde inlezen, …

Bottom-up aanpak

- Routine om één meetwaarde in te lezen
- Routines voor bepalen van elk kengetal
- Routine om kengetallen te verpakken in emailbericht
- Routine om emailbericht te versturen
- Routine om set meetwaarden in te lezen
- …

Nadelen

- Datumroutines bij inlezen en verzenden mogelijk dubbelop gemaakt
- Heel grote sets meetwaarden verwerken is problematisch: vergt (te) veel opslagruimte
- Opslaan was niet nodig (voorbarige keuze)
- Er moet dan veel veranderd worden in de software
- Alternatief: verwerken bij inlezen

Datastructuren

- Functionaliteit versus gegevens
- Algoritmen versus datastructuren
- Functionaliteit is niet de beste leidraad voor ontwerp
- Gegevens zijn betere leidraad
- Gegevens: opslag, operaties, gebruik

Abstracte Data Types

- Ontkoppel implementatiedetails (hoe opslag en operaties “in elkaar zitten”) zoveel mogelijk van gebruiksdetails
- Aparte functie voor elke operatie
- Gebruiker doet niets *direct* met de datastructuur; alles gaat via de aparte functies

Priority queue

- Beheer een *collectie objecten*, die elk een *waarde* hebben, met deze operaties:
  - Maak de collectie leeg
  - Ga na of de collectie leeg is
  - Voeg een gegeven object toe
  - Verwijder object met *minimum waarde* en retourneer dit object (als niet-leeg)
- Vgl. wachtrij met prioriteiten (= waarde)

Welke objecten?

- typedef … DATA …;
- typedef struct { int prio; DATA data; } OBJ;

Afwegingen

- Welke extra informatie bijhouden?
- Hoeveel opslagruimte kost het?
- Hoe beschrijf je de opslag handig?
- Hoe snel zijn operaties uit te voeren?

Ongesorteerd array

Na Clear						
Na Put(10)	10					
Na Put(42)	10	42				
Na Put(7)	10	42	7			
Na Put(3)	10	42	7	3		
Na Put(99)	10	42	7	3	99	
Na RemoveMin	10	42	7	99		

Eenvoudige aanpak

- Declareer array dat groot genoeg is om de grootst vóórkomende collectie te bevatten
- #define MAXN 1000000
- struct { int count; OBJ obj [MAXN]; } pq;
- $0 \leq$ pq.count $\leq$ MAXN (aantal in collectie)
- pq.obj[i] doet mee voor $0 \leq i <$ pq.count

Eenvoudige aanpak (2)

- Leeg maken: pq.count = 0;
- Is leeg?: pq.count == 0
- OBJ b toevoegen: pq.obj [pq.count] = b; pq.count = pq.count + 1;
- Minimum verwijderen (pq.count != 0):
  - bepaal i met minimale pq.obj[i].prio
  - pq.obj[i] afleveren
  - schuif pq.obj[j] met i < j naar pq.obj[j-1]

Minimum verwijderen

```
OBJ min_obj; // object met minimum prioriteit (resultaat)
int i_min; // index van minimum

// zoek het minimum
int min_prio = MAXPRIO; // minimum prioriteit tot nu toe
for ( int i = 0; i < pq.count; ++i ) {
  if ( pq.obj[i].prio < min_prio ) {
    i_min = i;
    min_prio = pq.obj[i].prio;
  }
}
// pq.obj[i_min].prio == min_prio is het minimum
min_obj = pq.obj [ i_min ];

// verwijder pq.obj [ i_min ], schuif de rest op
for ( int j = i_min + 1; j < pq.count; ++j ) {
  pq.obj [ j-1 ] = pq.obj [ j ];
}

pq.count = pq.count - 1;
```

Evaluatie

- Overhead bij opslag: één int count
- Leegmaken: één toekenning
- Leeg zijn: één vergelijking
- Toevoegen: twee toekenningen
- Minimum verwijderen: count stappen voor minimum bepalen, en nog eens *worst-case* count objectverplaatsingen

Gesorteerd array

Na Clear						
Na Put(10)	10					
Na Put(42)	42	10				
Na Put(7)	42	10	7			
Na Put(3)	42	10	7	3		
Na Put(99)	99	42	10	7	3	
Na RemoveMin	99	42	10	7		

Ietsje betere aanpak

- Als voorheen, met extra invariante relatie (kortweg: invariant):
- pq.obj is gesorteerd van hoog naar laag, d.w.z.
- Er geldt: pq.obj[i].prio ≥ pq.obj[i+1].prio voor alle i met 0 < i < pq.count

Ietsje betere aanpak (2)

- Leeg maken: pq.count = 0;
- Is leeg? pq.count == 0
- OBJ b toevoegen: zoek invoegplaats, schuif naar rechts, zet b op vrije plaats
- Minimum verwijderen (pq.count != 0):
  - pq.obj [pq.count-1] afleveren
  - pq.count = pq.count - 1;

Evaluatie

- Overhead bij opslag: één int count
- Leegmaken: één toekenning
- Leeg zijn: één vergelijking
- Toevoegen: log(count) stappen voor bepalen invoegplaats (binary search); count stappen (worst-case) voor opschuiven; één toekenning voor kopiëren
- Minimum verwijderen: twee toekenningen

Binary Search

- Zoek invoegplaats voor prioriteit p
- Halveer telkens kandidaatinterval [i, j)
- Invariant 1: $-1 \leq i < j \leq$ pq.count
- Invariant 2: pq.obj[i].prio $\geq$ p $>$ pq.obj[j].prio
- Afspraak: pq.obj[-1].prio = $-\infty$,
  pq.obj[pq.count].prio = ∞
- Start: i = -1; j = pq.count; // invarianten gelden nu
- Klaar als i+1== j: pq.obj[i].prio$\geq$p$>$pq.obj[i+1].prio
- Anders i+1 < j: m = (i + j)/2; // $0 \leq i < m < j \leq$ pq.count
  if (pq.obj[m].prio ≥ p) { i = m; }
  else /* p > pq.obj[m].prio */ { j = m; }

Binary Search in C

```
int bin_search ( int p )
  // pre: none
  // ret: index i met 0 <= i < pq.count en
           pq.obj[i].prio >= p > pq.obj[i+1].prio
{
  int i = -1, j = pq.count;
  // invariant 1: -1 <= i < j <= pq.count
  // invariant 2: pq.obj[i].prio >= p > pq.obj[j].prio

  while ( i+1 != j ) {
    int m = (i + j) / 2; // rounded down, 0 <= i < m < j <= pq.count
    if ( pq.obj[m].prio >= p ) {
      i = m;
    } else { // p > pq.obj[m].prio
      j = m;
    }
  }
  // i+1 == j en dus pq.obj[i].prio >= p > pq.obj[i+1].prio

  return i;
}
```

Ge‘link’te lijst

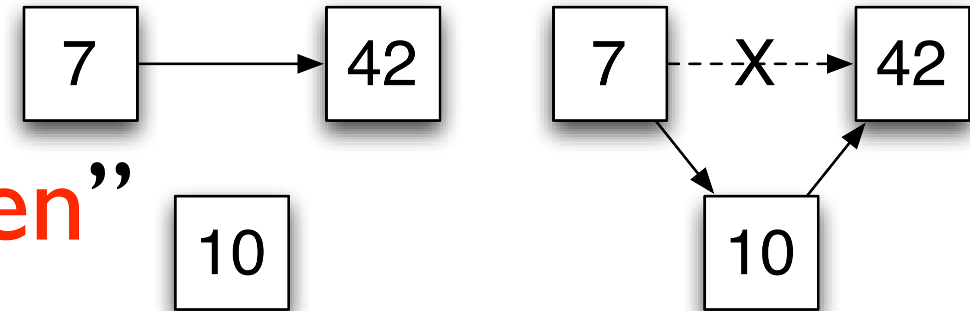
	i	0	1	2	3	4	5	first
Na Clear	prio[i]							
	next[i]							-1
Na Put(10)	prio[i]	10						
	next[i]	-1						0
Na Put(42)	prio[i]	10	42					
	next[i]	1	-1					0
Na Put(7)	prio[i]	10	42	7				
	next[i]	1	-1	0				2
Na Put(3)	prio[i]	10	42	7	3			
	next[i]	1	-1	0	2			3
Na Put(99)	prio[i]	10	42	7	3	99		
	next[i]	1	4	0	2	-1		3
Na RemoveMin	prio[i]	10	42	7		99		
	next[i]	1	4	0	-1	-1		2

Iets "betere" aanpak

- In 2e array gesorteerde volgorde bijhouden
- struct {…; int first; int next [MAXN]; } pq;
- pq.obj [pq.first] is minimum (pq.count>0)
- pq.obj [next[i]] is opvolger van pq.obj[i]
- pq.next[i] == –1 bij laatste object
- int free; // eerste van keten met vrije elementen

Iets “betere” aanpak (2)

- Leegmaken: pq.count = 0; first = free = -1;
- Is leeg?: pq.count == 0
- OBJ b toevoegen: “insorteren”
- Minimum verwijderen (pq.count != 0):
  minimum is pq.obj [pq.first].prio;
  pq.obj [pq.first] “vrijgeven”;
  pq.first = next [pq.first];
  pq.count = pq.count - 1;



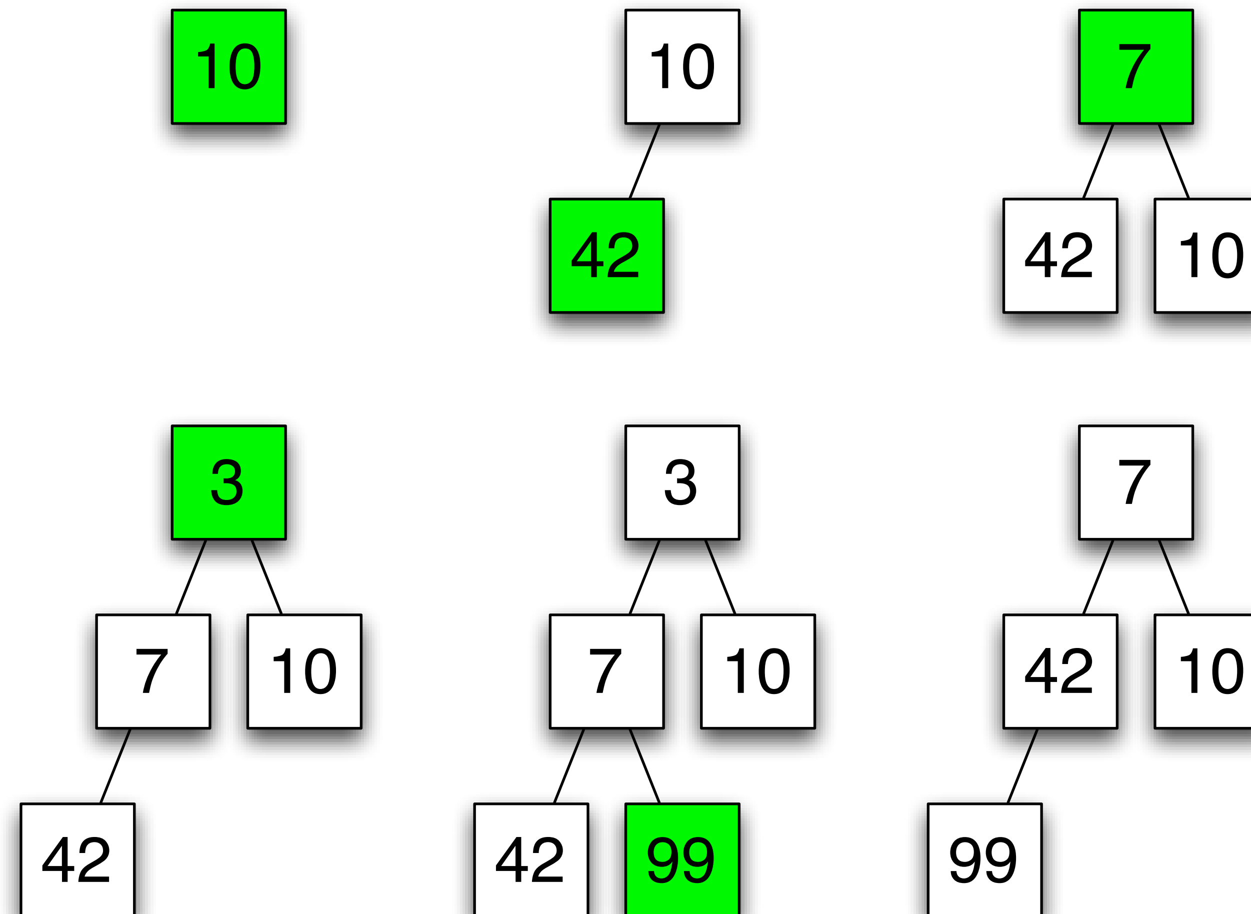
Insorteren

```
// bepaal vrij element om b in te plaatsen
int new; // doel: pq.obj [ new ] is vrij
if ( pq.free == -1 ) { // lijst met vrije elementen is leeg
  new = pq.count;
} else { // lijst is niet leeg, pak de eerste
  new = pq.free;
  pq.free = next [ pq.free ];
}
pq.obj [ new ] = b;

// pas pq.next aan
if ( pq.first == -1 || b.prio < pq.obj [ pq.first ].prio ) {
  // b moet vooraan komen
  next [ new ] = pq.first;
  pq.first = new;
} else { // pq.first != -1 && b.prio >= pq.obj [ pq.first ].prio
  // zoek index i waar OBJ b ingevoegd moet worden
  for ( int i = pq.first;
        pq.next[i] != -1 && pq.obj[ pq.next[i] ].prio <= b.prio;
        i = next[i] ) ;
  // i is eerste met pq.next[i] == -1 || pq.obj[pq.next[i]].prio > b.prio
  pq.next [ new ] = pq.next [ i ];
  pq.next [ i ] = new;
}

pq.count = pq.count + 1;
```

Evaluatie

- Overhead bij opslag: MAXN int
- Leegmaken: drie toekenningen
- Leeg zijn: één vergelijking
- Toevoegen: invoegplaats bepalen kost MAXN stappen (worst-case); invoegen kost vast aantal stappen
- Minimum verwijderen: vast aantal stappen

Betere aanpak: boom



Boom in array

	0	1	2	3	4	5
Na Clear						
Na Put(10)	10					
Na Put(42)	10	42				
Na Put(7)	7	42	10			
Na Put(3)	3	7	10	42		
Na Put(99)	3	7	10	42	99	
Na RemoveMin	7	42	10	99		

opvolgers van obj[i] zijn obj[2i+1], obj[2i+2]

voorganger van obj[i] is obj[(i-1)/2]

Betere aanpak: heap

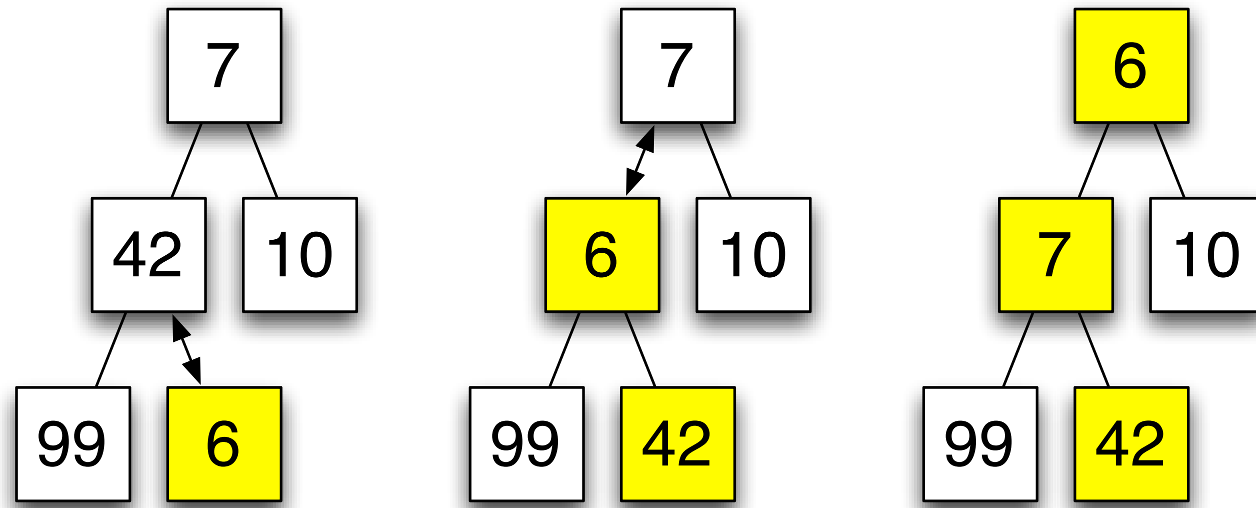
- Organiseer array obj[ ] als binaire heap: een binaire boom waarin obj[0] de wortel is, en obj[i] kinderen obj[2*i+1] en obj[2*i+2] heeft
- obj[i].prio $\leq$ obj[2*i+1], obj[2*i+2]
- dus obj[0].prio is minimum

Operaties

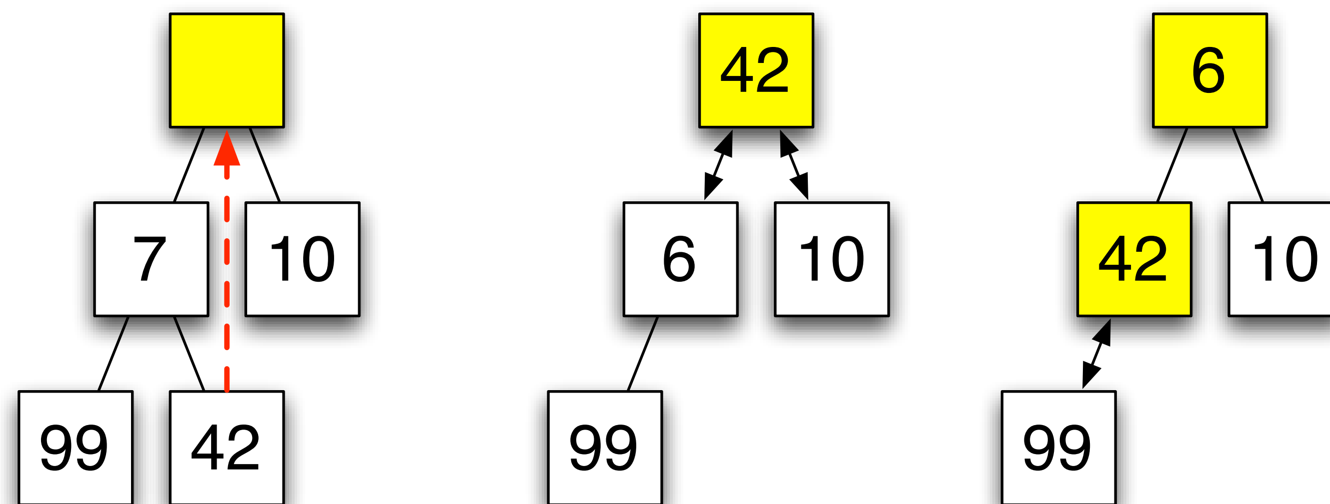
- Leegmaken: pq.count = 0;
- Is leeg?: pq.count == 0
- OBJ b toevoegen: pq.obj [pq.count] = b;
  gevolgd door "sift up" (logaritmisch)
- Minimum verwijderen (pq.count != 0):
  pq.obj[0] is minimum;
  pq.obj[0] = pq.obj [pq.count-1];
  gevolgd door "sift down" (logaritmisch)

Sift-up, Sift-down

- Sift-up:



- Sift-down



Afwegingen (herh.)

- Welke extra informatie bijhouden?
- Hoeveel opslagruimte kost het?
- Hoe beschrijf je de opslag handig?
- Hoe zijn operaties efficiënt uit te voeren?
- Pas op met plaatjes en kleine voorbeelden, want die zijn vaak misleidend en niet voldoende algemeen

Priority queue ADT

- Wat als je meer dan één priority queue nodig hebt?
- typedef struct
  { int count; OBJ obj [MAXN]; }
  PriorityQueue;
- Gebruiker doet niets direct met count en obj van PriorityQueue
- Dan is implementatie later eenvoudig te wijzigen zonder alle gebruik ook te hoeven aanpassen; ook nuttig i.v.m. verificatie

Priority queue ADT

- void clear_pq (PriorityQueue *pq);
- bool is_empty_pq (const PriorityQueue pq);
- void put_pq (PriorityQueue *pq, OBJ b);
- void remove_min_pq (PriorityQueue *pq, OBJ *b);
- PriorityQueue pq1, pq2;
- clear_pq (&pq1); put_pq (&pq1, b1);
  put_pq (&pq1, b2); remove_min_pq (&pq1, &b1);

Standaard ADTs voor collecties van objecten

- Stack, queue, priority queue
- Verzameling, ‘zak’ (Eng.: bag; met multipliciteiten)
- Dictionary, associative array
- Boom, graaf (netwerk van knopen en takken)

Implementatie technieken

- Arrays, eventueel met indexdoorverwijzingen in ander array
- Binary heap (impliciete doorverwijzingen)
- Pointers (pas op)
- Hash tables (voor gevorderden)