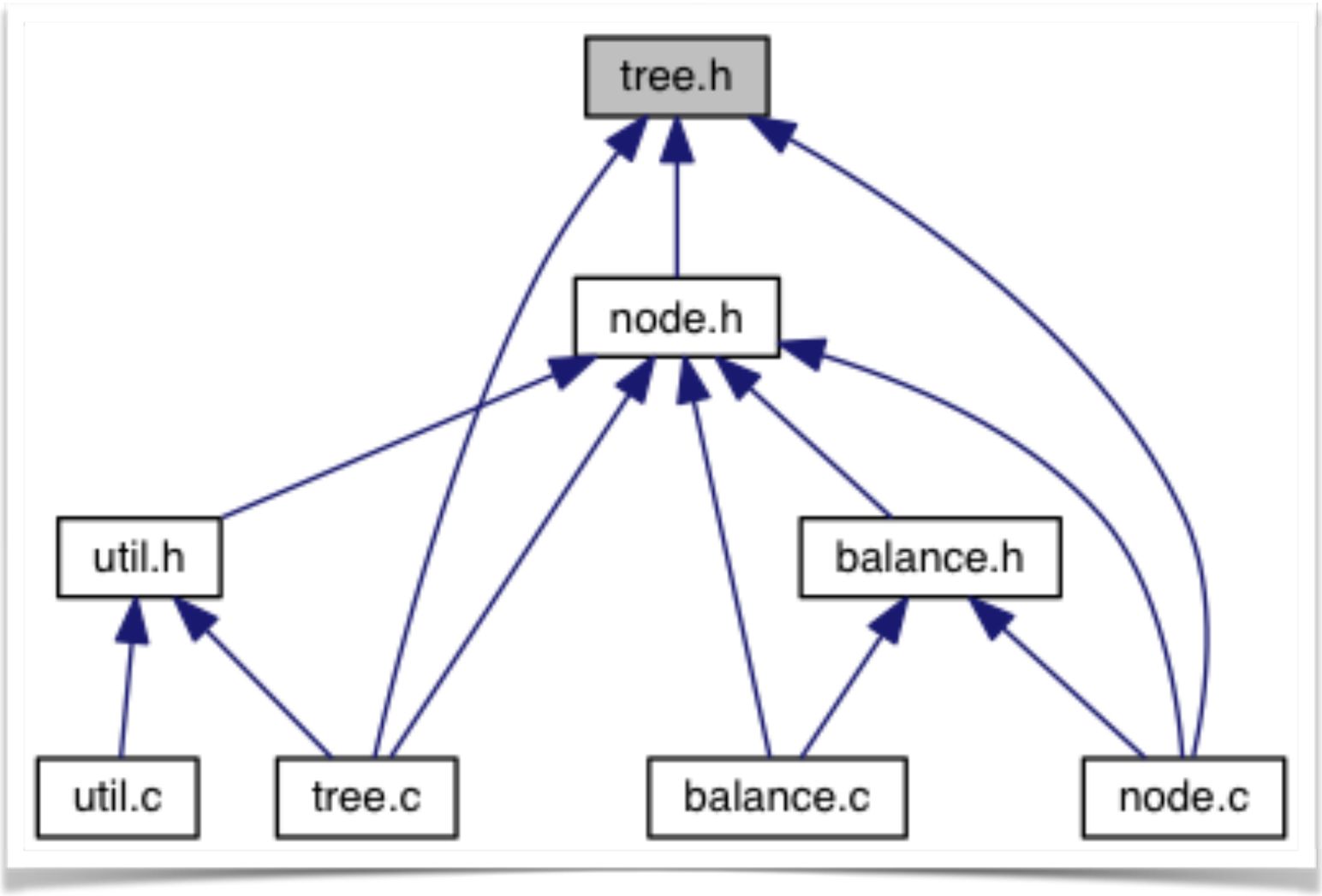
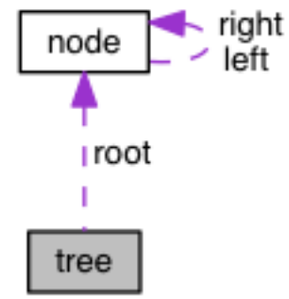
Avslutning Imperativ Programmering (i C)

Tobias Wrigstad

tobias.wrigstad@it.uu.se

En ”treemap” (från roliga timmen i fredags)





strukterna tree och node

Alla bilder tagna från doxygen

En ”treemap” (från roliga timmen i fredags)

Typdefinitioner

typedef struct tree **tree_s**
Trädet. Mer...

typedef int(* **cmp_f**)(void *, void *)
En jämförelsefunktion som används av trädet för att jämföra element. Mer...

typedef void(* **apply_f**)(void *, void *)
Signaturen för funktioner som skickas som argument till tree_traversal. Mer...

Egenuppräknande typer

enum **order** { **IN_ORDER**, **PRE_ORDER**, **POST_ORDER** }
Specificerar en traverseringordning. Mer...

Funktioner

tree_s *	**tree_new** (cmp_f key_compare_function, bool auto_balance)
void	**tree_delete** (tree_s *tree)
void	**tree_insert** (tree_s *tree, void *key, void *value)
void	**tree_traversal** (enum order order, tree_s *tree, apply_f operation, void *fst_arg)
uint32_t	**tree_depth** (tree_s *tree)
uint32_t	**tree_size** (tree_s *tree)
void *	**tree_get** (tree_s *tree, void *key)
void *	**tree_remove** (tree_s *tree, void *key)
char *	**tree_get_path_for_element** (tree_s *tree, void *key)
void *	**tree_get_element_for_path** (tree_s *tree, char *path)

En ”treemap” (från roliga timmen i fredags)

Datastrukturer

struct **node**

struct **key_value**

Typdefinitioner

typedef struct **node** **node_s**

Funktioner

node_s * **node_new** (void *k, void *v)

struct **key_value** **node_delete** (**node_s** *n)

Tar bort en nod ur minnet och returnerar en pekare till dess element. Mer...

uint32_t **node_depth** (**node_s** *n)

Returnerar subträdets djup. Mer...

uint32_t **node_size** (**node_s** *n)

Returnerar subträdets storlek. Mer...

void **node_traversal** (enum **order** **order**, **node_s** *n, **apply_f** f, void *arg, bool called_internally)

Traverserar subträdet. Mer...

struct **key_value** **unlink_smallest** (**node_s** **n_field)

void **node_insert** (**cmp_f** key_cmp, void *k, void *v, **node_s** **n, bool **balance**)

Stoppar in ett nytt element i trädet. Mer...

UU

En ”treemap” (från roliga timmen i fredags)

Egenuppräknande typer

```
enum  balance {
        BALANCED, LEFT_LEFT_HEAVY, LEFT_RIGHT_HEAVY, RIGHT_LEFT_HEAVY,
        RIGHT_RIGHT_HEAVY
      }
```

Funktioner

node_s *	rotate_left (node_s *n)
node_s *	flip_left (node_s *n)
node_s *	rotate_right (node_s *n)
node_s *	flip_right (node_s *n)
enum balance	balance_check (node_s *n) Kontrollerar hur välbalanserat ett subträd är. Mer...
void	balance_subtree (node_s ***ns, uint32_t ns_size)

Tree

- Uppdelat i 4 + 4 filer

  tree.h/c — det publika gränssnittet och dess implementation

  node.h/c — funktioner som manipulerar noder

  balance.h/c — balanseringsfunktioner

  util.h/c — hjälpfunktioner

- Tonvikten lagd på dokumentation av det **publika** gränssnittet, inte det privata

  Använder olika namnkonventioner publikt och privat

UU

Manuell minneshantering

- Konvention: matcha varje malloc med en free (helst i samma funktion!)
- För varje funktion som allokerar, skriv en som avallokerar, t.ex.

  tree_new & tree_delete

  node_new & node_delete
- Tips: konfigurera din editor för att synliggöra allokering
- Allokera små temporära data på stacken för att få automatisk minneshantering

  unlink_smallest, node_delete

Procedurell abstraktion

- Programmet är uppdelat i funktioner (procedurer) som anropar varandra i en anropskedja
- Varje funktionsnamn är valt för att ge en tydlig bild av vad den gör

  tree_insert

  tree_remove

  get_element_for_path

  tree_get

- I publik headerfil används extra långa parameternamn för dokumentation

UU

Informationsgömning

- Användaren av trädet vet inte hur träder är representerat internt — tree.h ger ingen sådan ledning

  struct tree definierad i tree.

- Men struct node definierad i node.h?

  Inte en del av det publika gränssnittet

  Underlättar testning (men många funktioner smutsar ned namnrymden ändå...)

Namnkonventioner

- Funktioner prefixade med tree_, node_ för tydlig tillhörighet

  Vissa funktioner behöver döpas om för att följa denna konvention!

- Korta namn på lokalt definierade symboler

  k = key

  v = value

  n = node

  t = tree

  Blir det enklare eller svårare att läsa?

Namnkonventioner

- Långa namn på globalt definierade symboler

  tree_get_element_for_path

  fält i struktar, t.ex. key och element

- Uppräkningsbara typer används av olika skäl på olika platser

  I tree.h för funktionalitet till klienten

  I node.c för att tydliggöra beteende (t.ex. node_traversal)

Defensiv programmering

- Använder inte det i någon större utsträckning i detta program
- Några undantag: offensiv programmering i switch-satser o.dyl.

  tree.c:138

  node.c:98

Makron

- Används för att gömma snårigheter vid en första läsning t.ex. Call och Val_or_whole

  Call(f, arg, ValOrWhole(n, whole));

  istället för

```
if (arg)
    f(arg, (whole ? n : n->element));
else
    f((whole ? n : n->element)), NULL);
```

Makron

- Används för att tydliggöra kodens innebörd, t.ex. No_left_subtree(n)

- Ibland nästlar vi makrodefinitioner i funktioner för att inte skapa ”global komplexitet”

Vissa makron kunde flyttas närmare användningsplatserna

Testbarhet

- Ännu inte helt beaktat
- Tillhandahåller vissa hjälpfunktioner för att tillåta att trädet inspekteras utan att abstraktionerna bryts
- Nodfunktionerna är (mestadels) testbara för sig

Läsbarhet

- Har kommenterats tidigare bl.a. namngivning
- Ibland används ”tabulering” för att lyfta fram kod som skall uttömma olika villkor, t.ex. i tree_remove

Imperativ programmering

- Många funktioner har skrivits på imperativ form utan rekursion — för vissa är anledningen mer pedagogisk än ”vettig” :)

  Se t.ex. node_insert som blir betydligt mer komplex då en särskild lista av föräldrar måste skapas

  Här glömde jag initialt free och hittade det endast med valgrind...

koden i kursrepot!

UU