Review of data types. Linear lists.

Chapter 4

2016

Data types

- The information processed by a computer program represents an abstraction of the real world. *Abstraction* means that some properties of the real objects are ignored and other are retained. Choosing the right abstraction is essential in solving a problem.
- In programming, constants, variables, expressions, functions and operators have associated a *data type*.
- The characteristics of a data type are:
  - a *set of values* a constant is member of, a variable can take, an expression can be evaluated to or a function/operator can generate.
  - The data type of a constant, variable, expression, function or operator can be inferred from its syntactic form or declaration.
  - Each operator and function have a *signature* which specifies the data types of its arguments and the data type of the result.
  - A data type assumes a certain *structuring level* of the information.

A classification of data types

- In a programming language we have *primitive data types* and *structured data types*.
- *Primitive date types* include: *standard types* (numbers, logical values, pointers) and *primitive user-defined types* (enumeration).
- *Structured types* are built from component types. If there is a single component type then it is called *base type*. Component types can be structured types as well, so we end up having a *hierarchy of data types*.
- A programming language provides *structuring methods* in order to define structured types. For example in C we have *arrays*, *structures* and *unions*.

Abstract data types - ADT

- ADT = a formal (mathematical) model of a data type. An ADT specifies a set of values (objects), a set of operators and their properties.
- ADTs can be understood easier by comparing them with procedures. The concept of procedure *generalizes* the notion of built-in function or operator. Another advantage of procedures is that they *encapsulate* parts of an algorithm in single processing statements.
- The use of procedures is called *procedural abstraction* and the use of ADTs is called *data abstraction*.
- An ADT *generalizes* the notion of data type and *encapsulates* a (new) data type.
- An ADT has a *definition* that states its mathematical model and *as many implementations*. Note however that:
  - Each implementation of an ADT must satisfy the definition of the ADT.
  - An ADT enforces the following usage discipline: using the ADT must be done only by calling the operators in the definition of the ADT. So hacks are not allowed ! In this way changing implementations preserves correctness.

Defining an ADT

1. **structure** Logic
2. **operators**
3. $false :\rightarrow Logic$
4. $true :\rightarrow Logic$
5. $and : Logic \times Logic \rightarrow Logic$
6. $or : Logic \times Logic \rightarrow Logic$
7. $not : Logic \times Logic \rightarrow Logic$
8. $if - then - else : Logic \times \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$
9. **axioms**
10. $false \neq true$
11. $not(false) = true$
12. $not(true) = false$
13. $and(true, X) = X$
14. $and(false, X) = false$
15. $or(X, Y) = not(and(not(X), not(Y)))$
16. $if - then - else(true, Y, Z) = Y$
17. $if - then - else(false, Y, Z) = Z$

An ADT is a tuple $\mathcal{T} = \langle \mathcal{D}, \mathcal{O}, \mathcal{A} \rangle$ such that:
i) $\mathcal{D}$ is the set of ADTs used in the definition of $\mathcal{T}$.
ii) $\mathcal{O}$ is the set of operators defined for $\mathcal{T}$.
iii) $\mathcal{A}$ is the set of axioms satisfied by the operators defined for $\mathcal{T}$.

Implementing an ADT

- An implementation of an ADT is a mapping that defines:
  - A representation of the objects of the ADT in terms of existing types.
  - The procedures corresponding to the operators of the ADT. The procedure definitions must satisfy the axioms of the ADT to be a *correct implementation* of the ADT.
- At the programming language level, an ADT implementation will correspond to:
  - A module in structured programming languages like C and Pascal.
  - A class in object oriented languages like C++ and Java.

Data type, abstract data type and data structure

- In programming terminology terms like *data type*, *abstract data type* and *data structure* are often misunderstood.
  - An *ADT* is a mathematical model (i.e. an abstraction), that is nevertheless very useful in the practice of programming.

  The focus is on the definition and on the semantics (i.e. its properties) of the external interface of the ADT.
  - A *data type*:
    - is an implementation of an ADT in a programming language or
    - it is provided by a programming language.

  The focus is on how the external interface of the data type is reflected in a particular implementation.
  - A *data structure* is an instance of a structured data type. By contrast, an elementary data is an instance of an unstructured data type. The focus is on the mechanisms of data structuring, rather than the external interface.

Dynamic sets

- In programming we are using very often sets that grow or shrink dynamically – *dynamic sets*.
- Often we need only the following operations on sets:
  - *insertion*
  - *removal* and
  - *search* (or *check for membership*).
- Such a dynamic set is often called a *dictionary* (using an analogy with the usual operations on a dictionary of words).

Elements of dynamic sets

- Usually set elements are implemented as complex structures (or objects) and are manipulated via pointers.
- Sometimes a field of an element x is a (unique) identification $key[x]$. Additional fields f are allowed, as well, $f[x]$.
- The keys might be taken from an ordered set $\mathcal{K}$. For example, if keys are strings then the order might be the lexicographical order defined on strings.
- The ordering of keys induces an ordering on the set elements:

$$x \le y \Leftrightarrow key[x] \le key[y]$$

Types of operations on dynamic sets

- Can be divided in two classes:

  – *Queries*: to get information about / from the set:
    - Check (get information about) the set and
    - Retrieve information from the set

  – *Updates*: to change the set:
    - Add
    - Remove and
    - Modify

  set elements.

Operations on dynamic sets

- SEARCH(S,k): it is a query that returns a pointer x to an element in S such that $key[x] = k$.
- INSERT(S,x): it is an update that adds to S the element pointed by x. This operations assumes that the element pointed by x has been initialized accordingly.
- REMOVE(S,x): it is an update that given a pointer to an element x in S removes x from S. Note that this operation uses a pointer to the element, not the key value of the element.
- MIN(S), MAX(S): they are queries that retrieve the minimum/maximum element from a totally ordered set S.
- NEXT(S,x), PREVIOUS(S,x): they are queries that retrieve the next / previous element greater / lower than x in S or NIL if x is the maximum/minimum element.

Definition of lists

- One of the most simple and most used ADT is the *linear list*. A linear list is a sequence of $n \geq 0$ elements X_1, ..., X_n called *nodes* having the same base type T.
- Essential structural properties of a linear list are:
  - If $n>0$ then X_1 is the first node and X_n is the last node.
  - If $1 < i < n$ then the i-th node X_i is preceded by X_{i-1} and succeeded by X_{i+1}. This property expresses the basis of *sequential access*, i.e. if we know the current node then we can easily get to its previous node and its next node.

Operations of lists

- To define lists as an ADT we must define a set of operators. Each set of operators defines a distinct ADT !
- There is a large variety of operators that can be defined on lists. Choosing the right set of operators is application dependent. Eg:
  - Provide the first or the last element of the list.
  - Compute the length of the list.
  - Append an element as the first or the last element of the list.
  - Check if the list is empty.
  - Remove the first or the last element of the list.
  - Decompose the list into the first element and the list that contains the rest of the elements.
  - Search for the first element with a given key.
  - a.o.

Linked lists

- A common way to implement lists uses *links*. A link is an address information to retrieve an element of the list.
- Links are implemented usually using *pointers*, but integer indices known as *cursors* can be used as well.
- In a *linked list* each element x contains a link to the next element $next[x]$ and possibly a link to the previous element $prev[x]$. If both links are present then the list is called *doubly-linked*, but if only $next[x]$ is present then the list is called *singly-linked*.
- If $prev[x] = NIL$ then x is the first element or the *head* of the list.
- If $next[x] = NIL$ then x is the last element or the *tail* of the list.
- A linked list can be *sorted* or not and can be *circular* or not.
- In a *sorted list* the elements are sorted according to the key order. The minimum element is the head and the maximum element is the tail.
- In a *circular list*, $next[tail] = head$ and $prev[head] = tail$. So the list has a ring shape. We must know at least one link to a node of the list !

Search in a linked list

The function LIST-SEARCH(L,k) searches for the first element encountered in L with key equal to k and returns a pointer to it. If there is no element in L with key k then the function returns NIL.

```
LIST-SEARCH(L, k)
1. x ← head[L]
2. while x ≠ NIL and key[x] ≠ k do
3.         x ← next[x]
4. return x
```

The function LIST-SEARCH takes $\Theta(n)$ time in the worst case to search for an element with a given key in a list with n elements.

Insertion into a linked list

The function LIST-INSERT(L,x) inserts an element x onto the first position in linked list L.

```
LIST-INSERT(L, x)
1. next[x] ← head[L]
2. if head[L] ≠ NIL then
3.     prev[head[L]] ← x
4. head[L] ← x
5. prev[x] ← NIL
```

The function LIST-INSERT takes $\Theta(1)$ time in the worst case to insert an element onto the first position in a list with n elements.

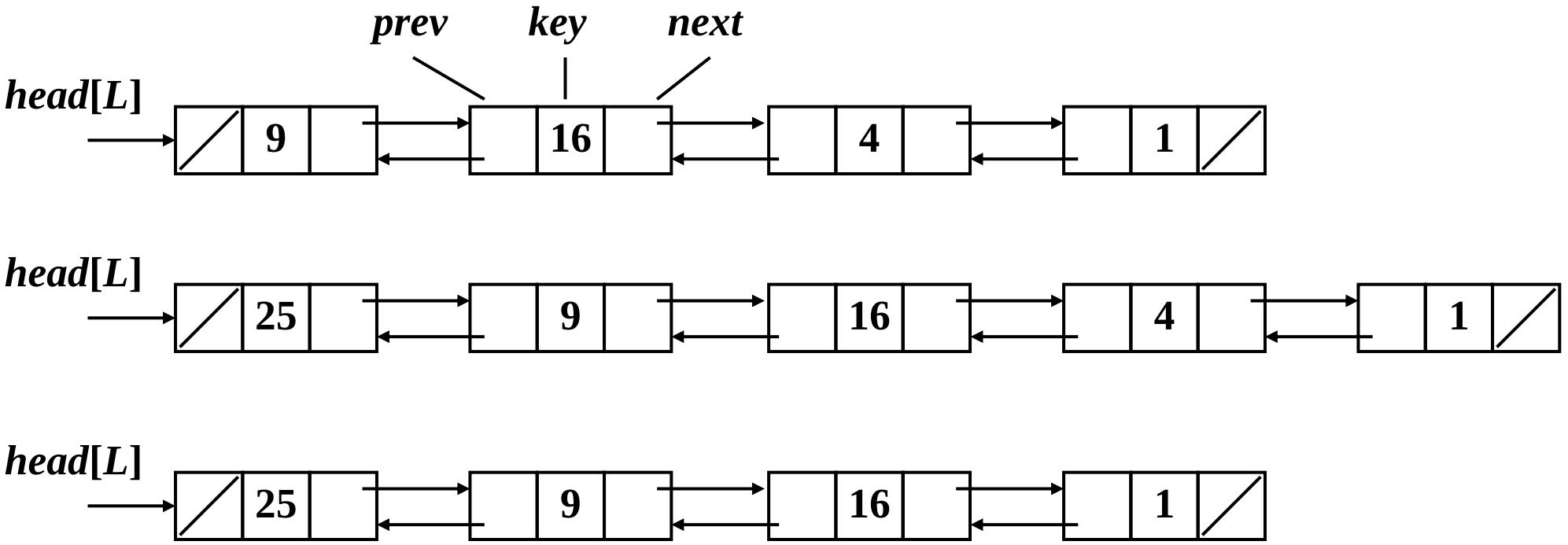
Removal from a linked list

The function LIST-REMOVE(*L*,*x*) removes an element *x* from a linked list *L*.

```
LIST-REMOVE(L, x)
1. if prev[x] ≠ NIL then
2.     next[prev[x]] ← next[x]
3. else
4.         head[L] ← next[x]
5. if next[x] ≠ NIL then
6.     prev[next[x]] ← prev[x]
```

The function LIST-REMOVE takes $\Theta(1)$ time in the worst case to remove an element from a list with *n* elements.
LIST-REMOVE can be combined with LIST-SEARCH to remove an element with a given key. This takes $\Theta(n)$ in the worst case.

Example



Consider a list L with elements 9, 16, 4, 1.
After executing LIST-INSERT(L,x) with $key[x] = 25$ we obtain the second list.
After the execution of LIST-REMOVE(L,x) with x pointing to element with $key[x]=4$ we obtain the third list.

Sentinels

The algorithm for removing an element would be simpler if we ignore the bounding conditions for the head and the tail of the list. To achieve this we can use sentinels.

$$\text{LIST-REMOVE}'(L, x)$$
$$1.\ next[prev[x]] \leftarrow next[x]$$
$$2.\ prev[next[x]] \leftarrow prev[x]$$

A *sentinel* is a dummy element *nil*[*L*] added in front of the list *L* that makes the list circular. So now:

- *next*[*nil*[*L*]] points to the head of the list and *prev*[*nil*[*L*]] points to the tail of the list.
- *head*[*L*] can be replaced with *next*[*nil*[*L*]].
- an empty list contains only the sentinel.

Search and insert with sentinels

LIST-SEARCH$'(L, k)$

1. $x \leftarrow next[nil[L]]$
2. **while** $x \neq nil[L]$ and $key[x] \neq k$ **do**
3. $\quad x \leftarrow next[x]$
4. **return** x

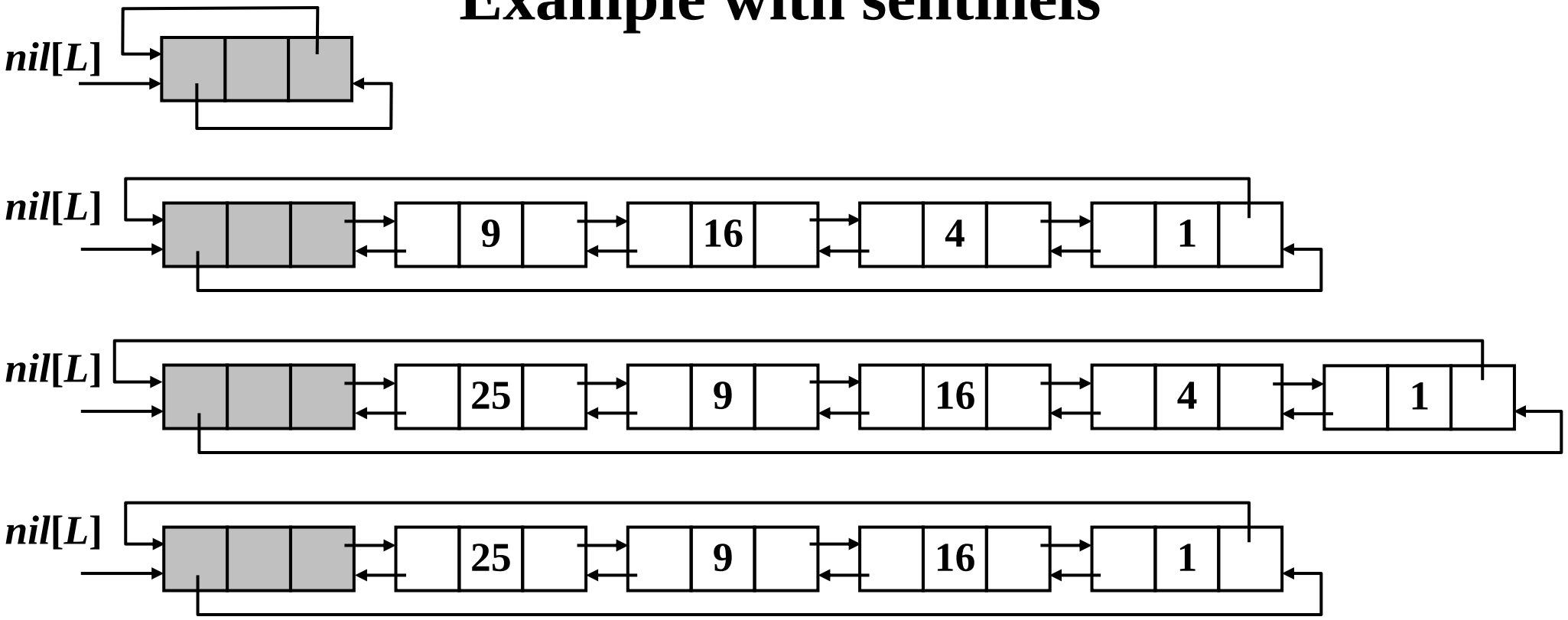
LIST-INSERT$'(L, x)$

1. $next[x] \leftarrow next[nil[L]]$
2. $prev[next[nil[L]]] \leftarrow x$
3. $next[nil[L]] \leftarrow x$
4. $prev[x] \leftarrow nil[L]$

The main advantage in using sentinels is the improvement of the readability of the code rather than efficiency in execution.

If in our program we have many small lists then the use of sentinels will result in a waste of storage.

Example with sentinels



The first list is the empty list.
Consider a list L with elements 9, 16, 4, 1. It is shown as the second list.
After executing LIST-INSERT'(L,x) with $key[x] = 25$ we obtain the third list.
After the execution of LIST-REMOVE'(L,x) with x pointing to element with $key[x]=4$ we obtain the third list.

Lists in C – the header file *list.h*

The program takes input lines of the form:
- `i` *number* – insert the number into the list
- `r` *number* – remove the number from the list
- `p` – print the list
- `x` – exit the program

The program has 2 modules and 3 source files:
- List module, files *list.h* and *list.c*
- Main module: file *main.c*

```
#ifndef LIST_H
#define LIST_H
#include <stdlib.h>

typedef struct list_node {
    struct list_node *next;
    struct list_node *prev;
    int key;
} ListNode;

typedef struct list {
    struct list_node *head;
} List;

ListNode* listSearch(List *l,int k);
void listInsert(List *l,ListNode *x);
void listRemove(List *l,ListNode *x);
#endif
```

Lists in C – the implementation file *list.c*

```
#include "list.h"
ListNode* listSearch(List *l,
  int k) {
  ListNode *x;
  x = l->head;
  while (x!=NULL && x->key!=k) {
    x = x->next;
  }
  return x;
}
void listInsert(List *l,
  ListNode *x) {
  x->next = l->head;
  if (l->head != NULL) {
    l->head->prev = x;
  }
  l->head = x;
  x->prev = NULL;
}
```

```
void listRemove(List *l,
  ListNode *x) {
  if (x->prev != NULL) {
    x->prev->next = x->next;
  }
  else {
    l->head = x->next;
  }
  if (x->next != NULL) {
    x->next->prev = x->prev;
  }
}
```

Lists in C – the main program *main.c*

```
#include <stdio.h>
#include "list.h"
void printList(List l) {
  ListNode *n;
  n = l.head;
  while (n != NULL) {
    printf("%3d",n->key);
    n = n->next;
  }
  printf("\n");
}
main() {
  List l;
  int k;
  char cmdLine[80];
  ListNode *n = NULL;
  l.head = NULL;
  gets(cmdLine);
  while (tolower(cmdLine[0])!='x') {
    switch(tolower(cmdLine[0])) {
    case 'i':
      sscanf(cmdLine+1,"%d",&k);
      n = (ListNode*)
        malloc(sizeof(ListNode));
      n->key = k;
      listInsert(&l,n);break;
    case 'r':
      sscanf(cmdLine+1,"%d",&k);
      n = listSearch(&l,k);
      if (n != NULL) {
        listRemove(&l,n);
      }
      break;
    case 'p':
      printList(l); break;
    }
    gets(cmdLine);
  } }
```

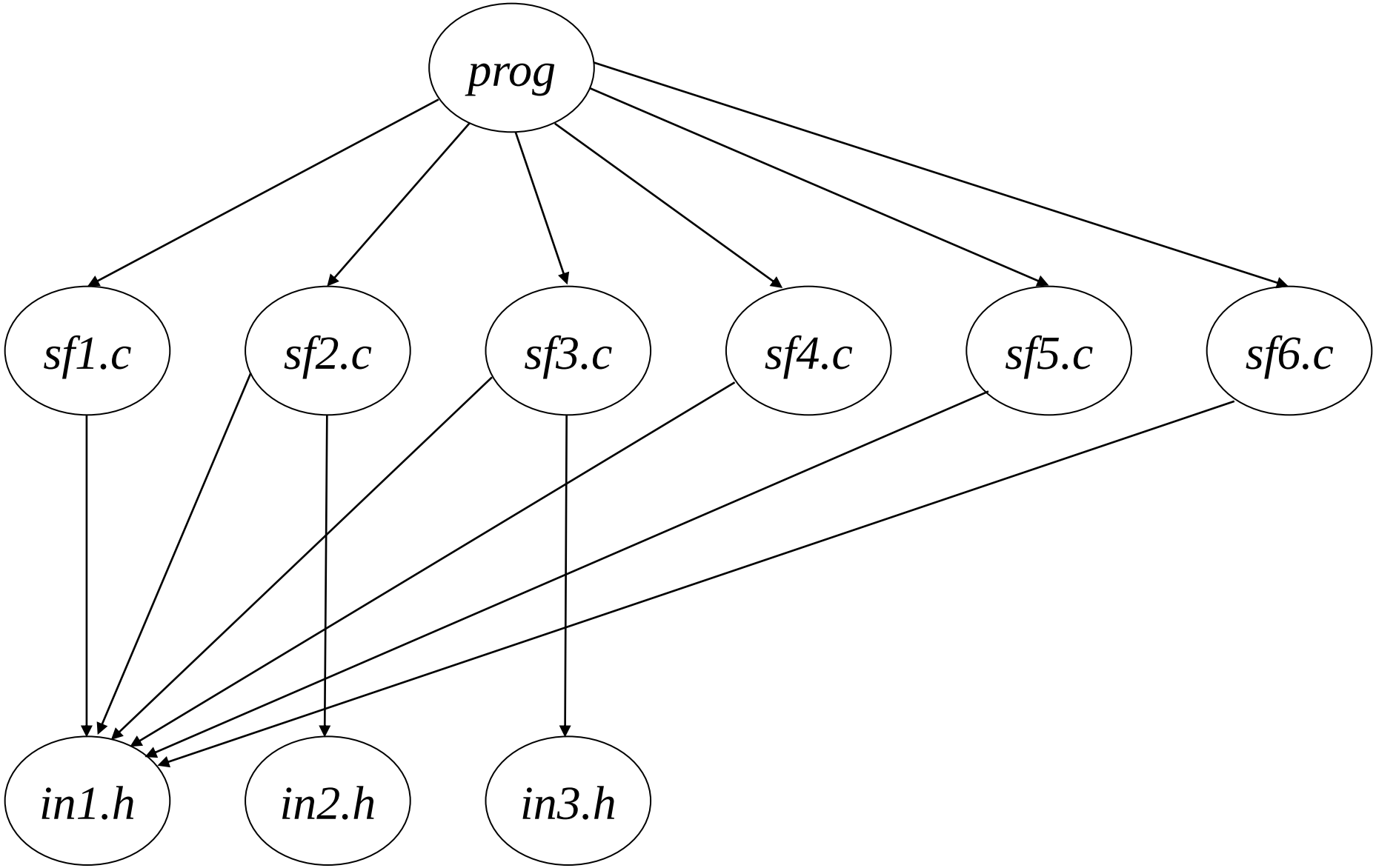
Modular Programming in C

- Non-trivial programs are composed of many source files:
  - Header files *.h* and
  - C source *.c* files.
- During the development process, other files may be used / produced:
  - Object *.o* (UNIX) or *.obj* (Windows) files
  - Executable files (for example *.exe* on Windows)
  - Library files (*.a* and *.so* on UNIX or *.lib* and *.dll* on Windows)
- Compiling all program source files (sometimes hundreds !) might consume quite a lot of time.
- At a certain time during program development, only a small number of source files are usually updated. So, there is no need to recompile all the source files whenever an update is made ⇒ *we must avoid recompilation of already compiled files !*

Project description files (*makefiles*)

- There are dependencies between file in the development process, foe example:
  - A *.c* file can depend on a *.h* file
  - A *.o* file can depend on a *.c* file
  - An executable file can depend on a *.o* file
- UNIX operating system (not Windows !) introduced the *make* utility to help with automating the management of dependencies between files (sources and binaries) generated during program development.
- The *make* utility needs a *project description file* in order to perform its tasks. This is called *makefile* of *Makefile*.
- Description file contains a set of rules that describe dependencies between output files and input files and the commands that need to be executed to create and/or update the output files.
- Let us consider a program composed of *sf1.c*, …, *sf6.c*, such that all include *in1.h*, *sf2.c* includes also *in2.h* and *sf3.c* includes also *in3.h*.

Example Modular Program



Example Makefile

```
prog : sf1.o sf2.o sf3.o sf4.o \
       sf5.o sf6.o
<T>gcc –o prog sf1.o sf2.o sf3.o sf4.o \
              sf5.o sf6.o

sf1.o : sf1.c in1.h
<T>gcc –c sf1.c
sf2.o : sf2.c in1.h in2.h
<T>gcc –c sf2.c
…
```

- We can avoid the repetition of file names *sf1.o* … *sf6.o* by introducing variables. For example, if *objects* is a variable, substitution of the variable with its content is specified as $(*objects*).

```
objects = sf1.o sf2.o sf3.o sf4.o \
       sf5.o sf6.o
prog : $(objects)
<T>gcc –o prog $(objects)

$(objects) : in1.h
sf2.o : in2.h
sf3.o : in3.h
```

Modular Programming in Python

- Non-trivial programs are composed of many source files:
  - A main file called *script*
  - Other source files called *modules*
- Many functions and classes are available in the built-in namespace. For example function *max*, class *list*, etc.
- Other useful functions and classes are organized into *modules* that can be *imported* from within a given program.
- Let us consider for example the *math* module. Definitions from a given module can be loaded into the current program (namespace) using *import* statement:

```
from math import e, sqrt  # imports data "e" and function "sqrt"
from math import *        # imports all the definitions
import math    # imports the module itself. Definitions are
               # accessed by fully-qualified name, for example
               # "math.e" or "math.sqrt(25)"
```

Linked Lists in Python

- A *list node* can be implemented:
  - As a Python list *x* with three elements: data *x*[0], next node *x*[1] and previous node *x*[2]
  - As a Python dictionary *x* with three elements: data *x*[‘key’], next node *x*[‘next’] and previous node *x*[‘prev’]
- A *list* can be implemented:
  - As an object representing the first element of the list (the list head)
  - As a Python dictionary *l* with one element *l*[‘head’] representing the first element of the list

Implementing Linked Lists in Python I

```
def node(x):
    return {'key':x,'next':None,'prev':None}

def list_search(l,k):
    x = l['head']
    while (x != None) and (x['key'] != k):
        x = x['next']
    return x

def list_insert(l,x):
    x['next'] = l['head']
    if l['head'] != None:
        l['head']['prev'] = x
    l['head'] = x
    x['prev'] = None
    return l
```

Implementing Linked Lists in Python II

```
def list_remove(l,x):
    if x['prev'] != None:
        x['prev']['next'] = x['next']
    else:
        l['head'] = x['next']
    if x['next'] != None:
        x['next']['prev'] = x['prev']
    return l
```

```
def list_write(l):
    x = l['head']
    while x != None:
        print(x['key']," ",end="")
        x = x['next']
    print("")
```

Avoid the ending newline with `end=""`.

Implementing Linked Lists in Python III

```
cmd = input()
l = {'head':None}
while cmd[0] != 'x':
    if cmd[0] == 'i':
        k = int(input())
        n = node(k)
        l = list_insert(l,n)
        print("i ",k)
    elif cmd[0] == 'r':
        k = int(input())
        n = list_search(l,k)
        if n != None:
            l = list_remove(l,n)
        print("r ",k)
    elif cmd[0] == 'p':
        print("p")
        list_write(l)
    cmd = input()
print("x")
```

```
C:\....>module_list.py <in.txt
i  2
i  4
i  7
i  8
p
8  7  4  2
r  4
p
8  7  2
i  8
p
8  8  7  2
r  9
p
8  8  7  2
i  9
r  8
p
9  8  7  2
i  8
p
8  9  8  7  2
x
```

in.txt

```
i
2
i
4
i
7
i
8
p
r
4
p
i
8
p
r
9
p
i
9
r
8
p
i
8
p
x
```

2016

Problems

1. The example program has a bug: it does not free the storage of the elements that are removed from the list. Fix this bug. Hint: use function *free*() from the C library.
2. Implement in C lists with sentinels following the model given in this lecture.
3. Modify the quick sort algorithm to work with doubly-linked lists and implement it in C.
4. Implement in C the operations INSERT, SEARCH and REMOVE for dictionaries. Assume that your keys are words.
5. Consider the union operation for sets. The operation takes two disjoint sets S_1 and S_2 as inputs and produces a set $S = S_1 \cup S_2$. Devise an O(1) algorithm for this operation using a suitable list structure. The sets S_1 and S_2 might be destroyed after the operation.

Problems (continuation)

6. Let us consider two sorted lists. Implement an algorithm to merge those two lists.
7. Let us suppose that sets are implemented using sorted lists. Implement efficient (linear time) algorithms for the:
   - Intersection of two sets
   - Union of two sets
8. *N* children play the well-known “ala-bala-portocala” game. They are placed on a circle and start counting from 1 to *k*. The *k*-th child is removed from the game. The process continues with the rest of the children, each time counting from 1 up to *k*, until all the children are removed from the game. Implement an efficient algorithm for printing out the order in which the children are removed from the game using a circular list. What is the execution time of your algorithm?