Review of data types.
Linear lists.

Chapter 4

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 data 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. \( \text{false} :\to \text{Logic} \)
4. \( \text{true} :\to \text{Logic} \)
5. \( \text{and} : \text{Logic} \times \text{Logic} \to \text{Logic} \)
6. \( \text{or} : \text{Logic} \times \text{Logic} \to \text{Logic} \)
7. \( \text{not} : \text{Logic} \times \text{Logic} \to \text{Logic} \)
8. \( \text{if} \rightarrow \text{then} \rightarrow \text{else} : \text{Logic} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T} \)

9. **axioms**
10. \( \text{false} \neq \text{true} \)
11. \( \text{not}(\text{false}) = \text{true} \)
12. \( \text{not}(\text{true}) = \text{false} \)
13. \( \text{and}(\text{true}, X) = X \)
14. \( \text{and}(\text{false}, X) = \text{false} \)
15. \( \text{or}(X, Y) = \text{not}(\text{and}(\text{not}(X), \text{not}(Y))) \)
16. \( \text{if} \rightarrow \text{then} \rightarrow \text{else}(\text{true}, Y, Z) = Y \)
17. \( \text{if} \rightarrow \text{then} \rightarrow \text{else}(\text{false}, Y, Z) = Z \)

An ADT is a tuple \( \mathcal{T} = (\mathcal{D}, \mathcal{O}, \mathcal{A}) \) 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.

- In programming terminology terms like data type, abstract data type and data structure are often misunderstood. To clarify and summarize:
  - An ADT is a mathematical model (i.e. an abstraction), that is nevertheless very useful in the practice of programming.
  - A data type is an implementation of an ADT in a programming language or it is provided by a programming language.
  - A data structure is an instance of a structured data type. By contrast, an elementary data is an instance of an unstructured data type.
Dynamic sets

- In programming we are using very often sets that grow or shrink dynamically – *dynamic sets*.

- Very often we only need the following operations on sets: *insertion, removal* and *search* (or *check for membership*). Such a dynamic set is called a *dictionary* (using an analogy with the usual operations on a dictionary of words).

- Usually set elements are implemented as complex structures (or objects) and are manipulated via pointers. Sometimes a field element is an identification *key*. Additional fields are allowed, as well. Also the keys might be taken from an ordered set. 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.

Operations on dynamic sets

- Can be divided in two classes:
  - *Queries*: to retrieve information from the set.
  - *Updates*: to modify the set.

- Operations:
  - SEARCH(S,k): query that returns a pointer \(x\) to an element in \(S\) such that \(\text{key}[x] = k\).
  - INSERT(S,x): update that adds to \(S\) the element pointed by \(x\). This operation assumes that the element pointed by \(x\) has been initialized accordingly.
  - REMOVE(S,x): 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): queries that retrieve the minimum/maximum element from a totally ordered set \(S\).
  - SUCCESSOR(S,x), PREDECESSOR(S,x): queries that retrieve the successor/predecessor element greater/lower than \(x\) in \(S\) or NIL if \(x\) is the maximum/minimum element.
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 \( T \).
• The 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 access its predecessor and its successor.
• To define lists as an ADT we must define a set of operators. Note that 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. Ex:
  – 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 successor element \( \text{next}[x] \) and possibly a link to the predecessor element \( \text{pred}[x] \). If both links are present then the list is called doubly-linked, but if only \( \text{next}[x] \) is present then the list is called singly-linked.
• If \( \text{pred}[x] = \text{NIL} \) then \( x \) is the first element or the head of the list.
• If \( \text{next}[x] = \text{NIL} \) 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, \( \text{next}[\text{tail}] = \text{head} \) and \( \text{pred}[\text{head}] = \text{tail} \). So the list has a ring shape.
Search in a linked list

The function \( \text{LIST-SEARCH}(L, k) \) 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.

\[
\text{LIST-SEARCH}(L, k) \\
1. \quad x \leftarrow \text{head}[L] \\
2. \quad \textbf{while} \ x \neq \text{NIL} \textbf{ and } \text{key}[x] \neq k \textbf{ do} \\
3. \quad \quad x \leftarrow \text{next}[x] \\
4. \quad \textbf{return} \ x
\]

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

\[
\text{LIST-INSERT}(L, x) \\
1. \quad \text{next}[x] \leftarrow \text{head}[L] \\
2. \quad \textbf{if} \ \text{head}[L] \neq \text{NIL} \textbf{ then} \\
3. \quad \quad \text{pred}[\text{head}[L]] \leftarrow x \\
4. \quad \text{head}[L] \leftarrow x \\
5. \quad \text{pred}[x] \leftarrow \text{NIL}
\]

The function \( \text{LIST-INSERT} \) 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** $\text{pred}[x] \neq \text{NIL}$ **then**
2. \hspace{1cm} $\text{next}[\text{pred}[x]] \leftarrow \text{next}[x]$
3. **else**
4. \hspace{1cm} $\text{head}[L] \leftarrow \text{next}[x]$
5. **if** $\text{next}[x] \neq \text{NIL}$ **then**
6. \hspace{1cm} $\text{pred}[\text{next}[x]] \leftarrow \text{pred}[x]$

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.

LIST-REMOVE'(L, x)
1. next[pred[x]] ← next[x]
2. pred[next[x]] ← pred[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 pred[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 ← next[nil[L]]
2. while x ≠ nil[L] and key[x] ≠ k do
3. x ← next[x]
4. return x

LIST-INSERT'(L, x)
1. next[x] ← next[nil[L]]
2. pred[next[nil[L]]] ← x
3. next[nil[L]] ← x
4. prd[x] ← 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 a lot of small lists then the use of sentinels will result in a waste of storage.
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*
Lists in C – the implementation file list.c

```c
#include "list.h"

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

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

Lists in C – the main program main.c

```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);
    }
}
```
Homework

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.

6. Devise an algorithm for merging two sorted lists and implement it in C.

Modular Programming

- Non-trivial programs are composed of many source files: `.h` or `.c` files. Compiling all program source files might take quite a lot of time.
- At a certain time during program development, only a small number of program source files are usually updated. Therefore, there is no need to recompile all the source files whenever an update is made.
- UNIX operating system (not Windows!) introduced the `make` utility to help the management of dependencies between files (sources and binaries) generated during program development.
- The `make` utility needs a description file in order to perform its tasks. This is called `makefile` or `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

![Diagram of 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