Eine Abstraktion für assoziative Arrays

- Assoziative Arrays (andere Namen: *hashes* und *dictionaries*) sind in den Grundoperationen vergleichbar mit normalen Arrays, abgesehen davon, daß
  - der Index-Typ beliebig sein kann (z.B. Zeichenketten) und
  - nicht notwendigerweise eine vollständige Ordnung für den Index-Typ existiert.
- In vielen Anwendungen sind assoziative Arrays sehr praktisch. So ist es z.B. bei einer Adreßdatenbank sinnvoll, wenn der Name als Index-Typ dient. Häufig wird der Index auch Schlüssel genannt.
- Manchmal sind auch Zugriffe über mehrere Schlüssel interessant. Bei Adreßdatenbanken könnte es beispielsweise nützlich sein, von der Telefon-Nummer oder der E-Mail-Adresse auf einen Datensatz zu schließen. Das sind dann sekundäre Schlüssel, die nicht notwendigerweise bei jedem Datensatz definiert sind.
- Das Hash-Verfahren ist ideal, um assoziative Arrays zu implementieren.

Eine Abstraktion für assoziative Arrays

- Folgendes wird bei einem konkreten assoziativen Array in Abhängigkeit des Datentyps benötigt:
  - Eine Hash-Funktion. Damit sie von der Tabellengröße unabhängig ist, kann sie einfach einen Wert vom Typ **INTEGER** zurückliefern. Die Implementierung des assoziativen Arrays kann dann den Wert in das Intervall $[0, n-1]$ abbilden.
  - Eine Operation, die zwei Schlüssel auf Identität prüft. Da die Hash-Funktion typischerweise nicht injektiv ist, müssen Schlüssel mit dem gleichen Hash-Wert voneinander unterschieden werden können.
  - Wie zuvor bei den binären sortierten Bäumen ist es sinnvoll, noch eine Operation hinzuzufügen, die überprüft, ob ein Objekt zu einem konkreten assoziativen Array "passt".
- Analog zu *SortedBinaryTrees* gibt es dann die Operationen *Insert*, *Delete* und *Lookup*.
- Da es jedoch keine Sortierung gibt, sind Operationen wie *Next*, *Prev* und eine Traverse in sortierter Reihenfolge nicht möglich. Stattdessen kann nur eine Iteration mit einer undefinierten Reihenfolge unterstützt werden.

Eine Abstraktion für assoziative Arrays

Hashes.od

```
TYPE
   HashValue = INTEGER;

   HashValProc =
      PROCEDURE (object: Objects.Object) : HashValue;
      (* return hash value of the key of this object *)

   HashableProc =
      PROCEDURE (object: Objects.Object) : BOOLEAN;
      (* return TRUE if hashval and equal may be invoked
         for object *)

   EqualProc =
      PROCEDURE (object1, object2: Objects.Object) : BOOLEAN;
      (* return TRUE if the keys of object1 and object2
         are identical *)

TYPE
   Hash = POINTER TO HashRec;
   HashRec = RECORD (Objects.ObjectRec) END;

PROCEDURE Create(VAR hash: Hash;
                 hashable: HashableProc;
                 hashval: HashValProc;
                 equal: EqualProc);
```

- Genauso wie *SortedBinaryTrees.Create* erhält *Hashes.Create* alle Prozeduren als Parameter, die von dem Typ des Schlüssels abhängen.

Eine Abstraktion für assoziative Arrays

Hashes.od

```
PROCEDURE Hashable(hash: Hash;
                   object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is compatible
      to those of hash
   *)

PROCEDURE Acceptable(hash: Hash;
                     object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is hashable and
      not yet present in hash
   *)

PROCEDURE Insert(hash: Hash; object: Objects.Object);
   (* precondition: object must be acceptable for hash
      adds object to hash
   *)

PROCEDURE Delete(hash: Hash; key: Objects.Object);
   (* precondition: key must be hashable for hash
      deletes object out of hash that has a key that
      is considered equal to key
   *)

PROCEDURE Lookup(hash: Hash; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be hashable for hash;
      stores into object that object of the hash
         that is identical to key if present;
      returns TRUE if one object has been found
   *)
```

Eine Abstraktion für assoziative Arrays

Hashes.od

```
PROCEDURE IterateHash(hash: Hash);
   (* start iteration of hash *)

PROCEDURE Next(hash: Hash;
               VAR object: Objects.Object) : BOOLEAN;
   (* store next object of iteration into object;
      note that the order of objects is undefined;
      returns FALSE on end of iteration
   *)
```

- *Hashes.IterateHash* startet einen Durchgang und *Hashes.Next* liefert die im assoziativen Array enthaltenen Objekte in einer vom Hash-Wert abhängigen Reihenfolge.

AddressManager.om

```
Hashes.IterateHash(db.hashByName);
WHILE Hashes.Next(db.hashByName, address) DO
   Addresses.PrettyPrint(Streams.stdout, address);
END;
```


Implementierung assoziativer Arrays

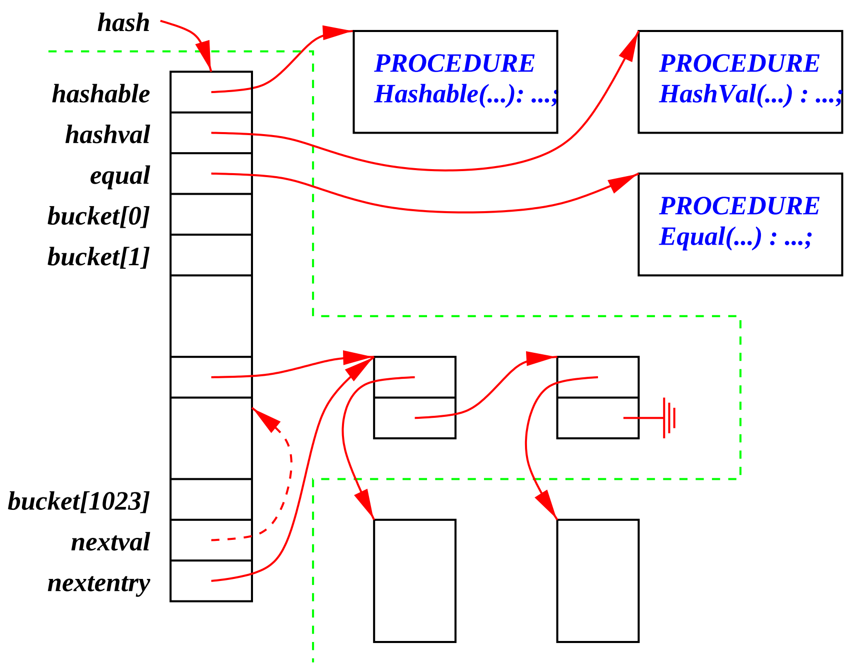
Hashes.om

```
CONST
   tabsize = 1024; logtabsize = 10;
TYPE
   Entry = POINTER TO EntryRec;
   EntryRec =
      RECORD
         object: Objects.Object;
         next: Entry; (* next entry with same hash value *)
      END;
   BucketTable = ARRAY tabsize OF Entry;
TYPE
   Hash = POINTER TO HashRec;
   HashRec =
      RECORD
         (Objects.ObjectRec)
         hashable: HashableProc;
         hashval: HashValProc;
         equal: EqualProc;
         bucket: BucketTable;
         (* for iterations: *)
         nextval: HashValue; nextentry: Entry;
      END;
```

- *hashable*, *hashval* und *equal* sind die Prozeduren, die bei *Hashes.Create* übergeben worden sind.

- *bucket* ist die *bucket table*, die mit den Hash-Werten indiziert wird und jeweils auf die Liste der Objekte mit dem gleichen Hash-Wert verweist.

Implementierung assoziativer Arrays



- Die gestrichelte (grüne) Linie kennzeichnet die Privatsphäre von *Hashes*

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE HashVal(hash: Hash;
                  object: Objects.Object) : HashValue;
   (* converts provided hash value into a hash value inside
      our range of [0..tabsize-1]
   *)
BEGIN
   RETURN
      SHORT(SYSTEM.LSH(1327217883 * hash.hashval(object),
                       logtabsize - 32))
END HashVal;

PROCEDURE Find(hash: Hash; key: Objects.Object;
               VAR hashval: HashValue;
               VAR entry, predecessor: Entry) : BOOLEAN;
BEGIN
   hashval := HashVal(hash, key);
   entry := hash.bucket[hashval]; predecessor := NIL;
   WHILE (entry # NIL) & ~hash.equal(entry.object, key) DO
      predecessor := entry; entry := entry.next;
   END;
   RETURN entry # NIL
END Find;
```

- *HashVal* bildet einen **INTEGER**-Wert, den die typabhängige Hash-Funktion liefert, in den Indexbereich der Hash-Tabelle ab.

- *Find* sucht nach *key* innerhalb von *hash*. Diese Prozedur wird von *Acceptable*, *Insert*, *Delete* und *Lookup* verwendet.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Create(VAR hash: Hash;
                 hashable: HashableProc;
                 hashval: HashValProc;
                 equal: EqualProc);
BEGIN
   NEW(hash);
   hash.hashable := hashable; hash.hashval := hashval;
   hash.equal := equal;
   hash.nextval := tabsize + 1; hash.nextentry := NIL;
END Create;

PROCEDURE Hashable(hash: Hash;
                   object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is compatible
      to those of hash
   *)
BEGIN
   RETURN hash.hashable(object)
END Hashable;

PROCEDURE Acceptable(hash: Hash;
                     object: Objects.Object) : BOOLEAN;
   (* return TRUE if the key of object is hashable and
      not yet present in hash
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   RETURN ~Find(hash, object, hashval, entry, predecessor)
END Acceptable;
```


Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Insert(hash: Hash; object: Objects.Object);
   (* precondition: object must be acceptable for hash
      adds object to hash
   *)
   VAR
      ok: BOOLEAN;
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   ok := ~Find(hash, object, hashval, entry, predecessor);
   ASSERT(ok);
   NEW(entry);
   entry.object := object;
   entry.next := hash.bucket[hashval];
   hash.bucket[hashval] := entry;
END Insert;
```

- Analog zu *SortedBinaryTrees* wird hier darauf bestanden, daß Schlüssel eindeutig sind.
- Genauso wie bei den Stapeln werden neu hinzukommende Einträge vor dem ersten Eintrag eingefügt.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Delete(hash: Hash; key: Objects.Object);
   (* precondition: key must be hashable for hash
      deletes object out of hash that has a key that
      is considered equal to key
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   IF Find(hash, key, hashval, entry, predecessor) THEN
      IF predecessor = NIL THEN
         hash.bucket[hashval] := entry.next;
      ELSE
         predecessor.next := entry.next;
      END;
   END;
END Delete;
```

- Das Löschen erfolgt analog zu einfachen linearen Listen, wobei *hashval* nur bestimmt, auf welcher Liste operiert wird.
- *predecessor* wird von *Find* auf den Vorgänger des zu löschenden Eintrags gesetzt. Dieser ist **NIL**, wenn der erste Eintrag zu löschen ist.

Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE Lookup(hash: Hash; key: Objects.Object;
                 VAR object: Objects.Object) : BOOLEAN;
   (* precondition: key must be hashable for hash;
      stores into object that object of the hash
         that is identical to key if present;
      returns TRUE if one object has been found
   *)
   VAR
      hashval: HashValue;
      entry, predecessor: Entry;
BEGIN
   IF Find(hash, key, hashval, entry, predecessor) THEN
      object := entry.object;
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Lookup;
```

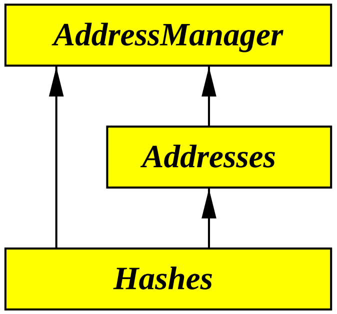
Implementierung assoziativer Arrays

Hashes.om

```
PROCEDURE IterateHash(hash: Hash);
   (* start iteration of hash *)
BEGIN
   hash.nextval := 0; hash.nextentry := hash.bucket[0];
END IterateHash;

PROCEDURE Next(hash: Hash;
               VAR object: Objects.Object) : BOOLEAN;
   (* store next object of iteration into object;
      note that the order of objects is undefined;
      returns FALSE on end of iteration
   *)
BEGIN
   IF hash.nextentry = NIL THEN
      WHILE (hash.nextval < tabsize) &
            (hash.bucket[hash.nextval] = NIL) DO
         INC(hash.nextval);
      END;
      IF hash.nextval >= tabsize THEN
         RETURN FALSE
      END;
      hash.nextentry := hash.bucket[hash.nextval];
      INC(hash.nextval);
   END;
   object := hash.nextentry.object;
   hash.nextentry := hash.nextentry.next;
   RETURN TRUE
END Next;
```


Anwendung: Eine kleine Adreßdatenbank



- Eine kleine Anwendung, die eine Adreßdatenbank realisiert, soll die Möglichkeiten von *Hashes* demonstrieren.
- Das Modul *Addresses* exportiert den Datentyp *Addresses.Address* zusammen mit zugehörigen Operationen zur Unterstützung von *Hashes* und zur Ein- und Ausgabe.
- Das Modul *AddressManager* offeriert eine interaktive Benutzerschnittstelle und dient als Hauptprogramm.

Der Datentyp *Addresses.Address*

Addresses.od

```
TYPE
   Address = POINTER TO AddressRec;
   AddressRec =
      RECORD
         (Objects.ObjectRec)
         name, email, address: ARRAY 128 OF CHAR;
         phone: ARRAY 32 OF CHAR;
      END;

PROCEDURE HashByName
            (object: Objects.Object) : Hashes.HashValue;
PROCEDURE HashByPhone
            (object: Objects.Object) : Hashes.HashValue;
PROCEDURE HashByEmail
            (object: Objects.Object) : Hashes.HashValue;

PROCEDURE EqualByName
            (object1, object2: Objects.Object) : BOOLEAN;
PROCEDURE EqualByPhone
            (object1, object2: Objects.Object) : BOOLEAN;
PROCEDURE EqualByEmail
            (object1, object2: Objects.Object) : BOOLEAN;

PROCEDURE Hashable(object: Objects.Object) : BOOLEAN;

PROCEDURE ReadFromFile(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
PROCEDURE WriteToFile(s: Streams.Stream;
                      address: Address) : BOOLEAN;
PROCEDURE ReadFromUser(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
PROCEDURE PrettyPrint(s: Streams.Stream; address: Address);
```


Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE HashByName
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).name)
END HashByName;

PROCEDURE HashByPhone
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).phone)
END HashByPhone;

PROCEDURE HashByEmail
            (object: Objects.Object) : Hashes.HashValue;
BEGIN
   RETURN HashByString(object(Address).email)
END HashByEmail;
```

- Für jede Art eines Zugriffs, sei es über den primären Schlüssel *name* oder einen der sekundären Schlüssel *phone* oder *email*, wird eine separate Hash-Funktion und eine separate Vergleichs-Operation benötigt.

Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE EqualByName
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).name = object2(Address).name
END EqualByName;

PROCEDURE EqualByPhone
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).phone = object2(Address).phone
END EqualByPhone;

PROCEDURE EqualByEmail
            (object1, object2: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object1(Address).email = object2(Address).email
END EqualByEmail;

PROCEDURE Hashable(object: Objects.Object) : BOOLEAN;
BEGIN
   RETURN object IS Address
END Hashable;
```


Der Datentyp *Addresses.Address*

```
thales$ am
: insert
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
Inserted.
: insert
Name:    Elisabeth Huber
Address: Neu-Ulm
Phone:   94721
E-Mail:  eh@huber.de
Inserted.
: dump addresses.db
: quit
thales$ cat addresses.db
Hans Maier~hans@maier.org~Ulm~25902
Elisabeth Huber~eh@huber.de~Neu-Ulm~94721
thales$
```

- Einfache Datensätze können so in einer Datei abgelegt werden, daß
  - jede Zeile einem Datensatz entspricht und
  - die Felder durch Feldtrenner voneinander getrennt sind.
- In der Ulmer Oberon-Bibliothek ist die Definition eines Feldtrenners über das Modul *StreamDisciplines* möglich.

Der Datentyp *Addresses.Address*

Addresses.om

```
PROCEDURE ReadFromFile(s: Streams.Stream;
                       VAR address: Address) : BOOLEAN;
   VAR
      ok: BOOLEAN;
BEGIN
   NEW(address);
   ok :=
      Read.FieldS(s, address.name) &
      Read.FieldS(s, address.email) &
      Read.FieldS(s, address.address) &
      Read.FieldS(s, address.phone);
   Read.LnS(s);
   RETURN ok
END ReadFromFile;

PROCEDURE WriteToFile(s: Streams.Stream;
                      address: Address) : BOOLEAN;
   VAR
      fieldsep: CHAR;
BEGIN
   StreamDisciplines.GetFieldSep(s, fieldsep);
   Write.StringS(s, address.name); Write.CharS(s, fieldsep);
   Write.StringS(s, address.email); Write.CharS(s, fieldsep);
   Write.StringS(s, address.address);
   Write.CharS(s, fieldsep);
   Write.StringS(s, address.phone); Write.LnS(s);
   RETURN s.errors = 0
END WriteToFile;
```


Die Adreßdatenbank

```
thales$ am
: load addresses.db
: phone 25902
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
: email eh@huber.de
Name:    Elisabeth Huber
Address: Neu-Ulm
Phone:   94721
E-Mail:  eh@huber.de
: name "Hans Maier"
Name:    Hans Maier
Address: Ulm
Phone:   25902
E-Mail:  hans@maier.org
: quit
thales$
```

- Datensätze sollen aufgrund des Namens (primärer Schlüssel), der Telefon-Nummer oder der E-Mail-Adresse (sekundäre Schlüssel) gefunden werden.
- Entsprechend werden drei assoziative Arrays benötigt...

Die Adreßdatenbank

AddressManager.om

```
TYPE
   Database =
      RECORD
         hashByName, hashByPhone, hashByEmail: Hashes.Hash;
         (* names act as primary keys (i.e. we expect them
            to be unique) but we support phone numbers
            and email addresses as secondary indices as well;
            if secondary keys are not unique we just keep a
            reference to the first object we have seen with
            that key
         *)
      END;

PROCEDURE CreateDB(VAR db: Database);
   (* create the three hashes associated with db *)
BEGIN
   Hashes.Create(db.hashByName, Addresses.Hashable,
      Addresses.HashByName, Addresses.EqualByName);
   Hashes.Create(db.hashByPhone, Addresses.Hashable,
      Addresses.HashByPhone, Addresses.EqualByPhone);
   Hashes.Create(db.hashByEmail, Addresses.Hashable,
      Addresses.HashByEmail, Addresses.EqualByEmail);
END CreateDB;
```

- Für den primären und für die beiden sekundären Schlüssel wird je ein assoziatives Array unterhalten.

Die Adreßdatenbank

AddressManager.om

```
PROCEDURE Insert(db: Database;
                 address: Addresses.Address) : BOOLEAN;
   (* insert address into db (if it is acceptable,
      otherwise RETURN FALSE) by storing it into
      db.hashByName (accessible by the primary key) and to
      db.hashByPhone and hashByEmail if these fields are
      non-empty and these keys are not yet used
   *)
BEGIN
   IF Hashes.Acceptable(db.hashByName, address) THEN
      Hashes.Insert(db.hashByName, address);
      IF (address.phone # "") &
            Hashes.Acceptable(db.hashByPhone, address) THEN
         Hashes.Insert(db.hashByPhone, address);
      END;
      IF (address.email # "") &
            Hashes.Acceptable(db.hashByEmail, address) THEN
         Hashes.Insert(db.hashByEmail, address);
      END;
      RETURN TRUE
   ELSE
      RETURN FALSE
   END;
END Insert;
```

- Wenn ein neuer Eintrag hinzukommt, ist er in jeden der assoziativen Arrays einzufügen (soweit jeweils die Schlüssel gegeben sind).
- Da *Hashes* auf die Eindeutigkeit der Schlüssel besteht, kann pro sekundärem Schlüssel jeweils nur ein Eintrag erfolgen.

Die Adreßdatenbank

AddressManager.om

```
PROCEDURE Delete(db: Database; address: Addresses.Address);
   (* delete address from db.hashByName and
      the other two hashes if present
   *)
   VAR
      otherAddress: Addresses.Address;
BEGIN
   Hashes.Delete(db.hashByName, address);
   IF (address.phone # "") &
         Hashes.Lookup(db.hashByPhone,
            address, otherAddress) &
         (address = otherAddress) THEN
      Hashes.Delete(db.hashByPhone, address);
   END;
   IF (address.email # "") &
         Hashes.Lookup(db.hashByEmail,
            address, otherAddress) &
         (address = otherAddress) THEN
      Hashes.Delete(db.hashByEmail, address);
   END;
END Delete;
```

- Bei *Delete* sind ebenfalls alle assoziativen Arrays zu berücksichtigen.
- Dabei ist hier zu achten, daß wir bei den sekundären Indizes nur die Einträge löschen, die von dem gleichen Datensatz stammen.

Literaturhinweise zu Datenstrukturen auf Basis von Hash-Verfahren

Zur weiteren Vertiefung sind folgende Texte geeignet:

- Donald E. Knuth, "The Art of Computer Programming", Band 3 "Sorting and Searching", Abschnitt 6.4 über "Hashing". Dieser Abschnitt trug wesentlich zu den einführenden Seiten dieses Kapitels bei.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", Kapitel 12 "Hash Tables".
- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, "Data Structures and Algorithms", Kapitel 4 "Basic Operations on Sets" und speziell Abschnitt 4.7.
  Hier wird gezeigt, wie auf Basis von Hash-Tabellen eine Abstraktion für Mengen realisiert werden kann.