Nispeten daha karmaşık programlama problemlerinden biri, bir seyrek dizinin gerçeklenmesidir. *Seyrek dizi*, dizinin tüm öğelerinin esasen kullanılmadığı, mevcut olmadığı ya da gerekmediği bir dizidir. Seyrek diziler, şu koşulların her ikisi ile karşılaşıldığında çok yararlıdır: Bir uygulamanın gereksinim duyduğu dizinin büyüklüğü oldukça büyüktür (muhtemelen mevcut belleği aşmaktadır) ve tüm dizi konumları kullanılmayacaktır. Dolayısıyla, bir seyrek dizi genellikle içine az değer yerleştirilmiş büyük bir dizidir. Göreceksiniz ki, seyrek dizileri uygulayabilmek için birkaç yöntem mevcuttur. Başlamadan önce, çözüm olarak seyrek dizilerin tasarlandığı bir problemi inceleyelim.

Seyrek Dizilere Duyulan Gereksinimi Anlamak

Seyrek dizilere duyulan gereksinimi anlamak için aşağıdaki iki hususu ele alın:

- Normal bir C dizisi deklare ettiğinizde, dizinin gerektirdiği tüm bellek, dizi var olduğu anda tahsis edilir.
- Büyük diziler – özellikle çok boyutlu diziler – çok fazla miktarda bellek tüketebilir.

Dizi için gerekli belleğin dizi oluşturulurken ayrılması, programınızın içinde deklare edebileceğiniz en büyük dizinin mevcut bellek ile (kısmen) sınırlı olması demektir. Bilgisayarınızın fiziksel sınırları içinde kalan bir diziden daha büyüğüne gerek duyuyorsanız, diziyi desteklemek için başka bazı mekanizmalar kullanmalısınız. (Söz gelişi, tamamen dolu büyük bir dizi genellikle bir tür sanal bellek kullanır.) Büyük dizi belleğe sığacak olsa bile, onu deklare etmek her zaman iyi bir fikir olmayabilir. Çünkü, büyük dizi tarafından tüketilen bellek, programınızın geri kalanı tarafından ya da sistem içinde çalışmakta olan diğer prosesler tarafından kullanılamaz. Bu, programınızın ya da bilgisayarın genel performansını azaltabilir. Tüm dizi konumlarının gerçekte kullanılmayacağı durumlarda büyük bir dizi için bellek ayırmak özellikle sistem kaynaklarını savurganca kullanmaktır.

Büyük ve az içeriğe sahip dizilerin neden olduğu problemleri çözmek için, birkaç seyrek dizi tekniği icat edilmiştir. Tüm seyrek dizi teknikleri ortak bir noktayı paylaşırlar: Dizi öğeleri için yalnızca gerektiğinde bellek ayırırlar. Dolayısıyla, seyrek dizinin avantajı, yalnızca gerçekten kullanılan öğeler için bellek gerektirmesidir. Bu, belleğin geri kalanını diğer kullanımlar için serbest bırakır. Ayrıca, son derece büyük – normalde sistem tarafından izin verilenden daha büyük – dizilerin kullanılmasına olanak verir.

Seyrek dizilerin işlenmesini gerektiren sayısız uygulama örneği mevcuttur. Bunların çoğu matris işlemlerine ya da yalnızca alanlarının uzmanlarınca kolayca anlaşılan bilimsel ve mühendislik problemlerine uygulanır. Bununla birlikte, çok tanıdık bir uygulama da tipik olarak seyrek dizileri kullanmaktadır: Elektronik tablo (spreadsheet) programı. Ortalama bir elektronik tablo matrisi çok büyük olmakla birlikte, aslında matrisin yalnızca bir bölümü bir anda kullanımdadır. Elektronik tablolar her konumda ilintili formülleri, değerleri ve karakter katarlarını tutmak amacıyla matris kullanırlar. Bir seyrek dizi kullanıldığında, her öğenin depolanması için, gerektikçe boş bellek havuzundan yer ayrılır. Gerçekte dizi öğelerinin yalnızca küçük bir bölümü kullanımda olduğu için, dizi (yani, elektronik tablo) yalnızca fiilen kullanılmakta olan o hücreler için bellek gerektiriyor olmakla birlikte çok büyük görünebilir.

Bu bölümde iki terim tekrarlanarak kullanılacaktır: *Mantıksal dizi* ve *fiziksel dizi*. Mantıksal dizi, sistemde mevcutmuş gibi düşündüğünüz dizidir. Örneğin, bir elektronik tablo 1.000 x 1.000 boyutuna sahipse, bu matrisi destekleyen mantıksal dizinin de boyutları 1.000 x 1.000'dir. Ancak bu dizi, bilgisayarın içinde fiziksel olarak mevcut değildir. Fiziksel dizi ise bilgisayarın içinde liilen mevcut olan dizidir. Bu nedenle, bir elektronik tablo matrisinin yalnızca 100 öğesi kullanılıyorsa, fiziksel dizi yalnızca o 100 oğe için bellek gerektirir. Bu bölümde geliştirilen seyrek dizi teknikleri mantıksal ve fiziksel diziler arasında bağlantı kurmaktadırlar.

Bu bölümde bir seyrek dizi oluşturmak için dört farklı teknik incelenmektedir: Bağlı liste, ikili ağaç, bir işaretçi dizisi ve hashing. Aslında bir elektronik tablo programı geliştirilmemiş olmakla birlikte, tüm örnekler Şekil 23.1'de gösterilen şekilde düzenlenen bir elektronik tablo matrisi ile bağlantılıdır. Şekilde X, B2 hücresinde konumludur.

Bağlı Liste Şeklinde Bir Seyrek Dizi

Bir seyrek diziyi bir bağlı liste kullanarak uyguladığınızda, ilk yapmanız gereken şey aşağıdaki unsurları içeren bir yapı oluşturmaktır.

- Depolanmakta olan veri
- Bu verinin dizi içindeki mantıksal konumu
- Önceki ve sonraki öğelere bağlantılar

ŞEKİL 23.1: Basit bir elektronik tablonun düzenlenişi.

Her yeni yapı, öğeleri dizi indeksi temel alınarak sıralı bir düzende eklenmek suretiyle listeye yerleştirilir. Dizi, bağlantılar izlenerek erişilir.

Örneğin, aşağıdaki yapıyı bir elektronik tablo programında bir seyrek dizinin temeli olarak kullanabilirsiniz:

```
struct cell {
  char cell_name[9];  /* hucre adi, ornegin A1, B34 */
  char  formula[128]; /* bilgi, ornegin 10/B2 */
  struct cell *next;  /* bir sonraki kaydi gosteren isaretci */
  struct cell *prior; /* bir onceki kaydi gosteren isaretci */
} ;
```

`cell_name` alanı, A1, B34 ya da Z19 gibi bir hücre adı içeren bir karakter katarını tutar. `formula` karakter katarı, elektronik tablo konumunun her birine atanan formülü (veriyi) tutar.

Bütün bir elektronik tablo, bir örnek olarak kullanmak için haddinden fazla büyük olacaktır. Onun yerine, bu bölümde bağlı liste şeklinde bir seyrek diziyi destekleyen temel fonksiyonlar incelenmektedir. Hatırlarsanız, bir elektronik tablo programını uygulamak için birçok yöntem mevcuttur. Buradaki veri yapıları ve rutinler yalnızca seyrek dizi tekniklerinin birer örneğidir.

Aşağıdaki global değişkenler bağlı liste dizisinin başına ve sonuna işaret ederler:

```
struct cell *start = NULL; /* listenin ilk ogesi */
struct cell *last = NULL;  /* listenin son ogesi */
```

Birçok elektronik tabloda bir hücreye bir formül girdiğimizde, aslında seyrek dizi içinde yeni bir öğe oluşturuyorsunuz demektir. Eğer elektronik tablo bir bağlı liste kullanıyorsa, o yeni hücre Bölüm 22'de geliştirilen `dls_store()`'a benzer bir fonksiyon aracılığıyla listenin içine eklenir. Listenin hücre adına göre sıralı olduğunu hatırlayın. Yani, A12 A13'ten önce gelir vs.

```
/* Hucreleri sirali duzende depolar. */
void dls_store(struct cell *i, /* eklenecek yeni hucreyi gosteren isaretci */
               struct cell **start,
               struct cell **last)
{
  struct cell *old, *p;

  if(!*last) { /* listenin ilk ogesi */
    i->next = NULL;
    i->prior = NULL;
    *last = i;
    *start = i;
    return;
  }

  p = *start; /* start, listenin basindadir */

  old = NULL;
  while(p) {
    if(strcmp(p->cell_name, i->cell_name) < 0){
      old = p;
      p = p->next;
    }
    else {
      if(p->prior) { /* ortadaki ogedir */
        p->prior->next = i;
        i->next = p;
        i->prior = p->prior;
        p->prior = i;
        return;
      }
      i->next = p; /* yeni ilk ogedir */
      i->prior = NULL;
      p->prior = i;
```

```
      *start = i;
      return;
    }
  }
  old->next = i; /* sona yerleştir */
  i->next = NULL;
  i->prior = old;
  *last = i;
  return;
}
```

Burada `i` parametresi, eklenecek yeni hücreyi gösteren bir işaretçidir. `start` ve `last` parametreleri sırasıyla, listenin başını ve sonunu işaret eden işaretçileri gösteren işaretçilerdir.

`deletecell()` fonksiyonu – aşağıda yer almaktadır – adı fonksiyona argüman olarak aktarılan hücreyi listeden çıkartır.

```
void deletecell(char *cell_name,
                struct cell **start,
                struct cell **last)
{
  struct cell *info;

  info = find(cell_name, *start);
  if(info) {
    if(*start==info) {
      *start = info->next;
      if(*start) (*start)->prior = NULL;
      else *last = NULL;
    }
    else {
      if(info->prior) info->prior->next = info->next;
      if(info != *last)
        info->next->prior = info->prior;
      else
        *last = info->prior;
    }
    free(info); /* belleği sisteme döndür */
  }
}
```

Bir bağlı liste şeklindeki seyrek diziyi desteklemek için gerek duyacağınız son fonksiyon, herhangi bir spesifik hücreyi bulan `find()`'dir. Fonksiyon, her öğenin konumunu bulmak için doğrusal arama gerektirir. Bölüm 21'de gördüğünüz gibi, doğrusal bir aramada ortalama karşılaştırma sayısı $n/2$'dir. Burada n, listedeki öğe sayısıdır. `find()` şu şekildedir:

```
struct cell *find(char *cell_name, struct cell *start)
{
  struct cell *info;

  info = start;
  while(info) {
```

```
    if(!strcmp(cell_name, info->cell_name)) return info;
    info = info->next; /* bir sonraki hucreyi al */
  }
  printf("Cell not found.\n");
  return NULL; /* bulunamadi */
}
```

Bağlı Liste Yaklaşımının Analizi

Seyrek diziler için bağlı liste yaklaşımının başlıca avantajı, belleğin verimli biçimde kullanılmasıdır; bellek yalnızca dizinin gerçekten bilgi içeren öğeleri için kullanılır. Ayrıca, bu yöntemin uygulanması da basittir. Ancak, bunun temel bir dezavantajı söz konusudur. Listedeki hücrelere erişmek için doğrusal arama kullanılmalıdır. Üstelik, depolama rutini, yeni bir hücreyi listeye eklemek için doğrusal arama kullanarak uygun konumu bulur. Bu problemleri seyrek diziyi destekleyen ikili ağaç kullanarak çözebilirsiniz. Sırada bu gösterilmiştir.

Seyrek Dizilere İkili Ağaç Yaklaşımı

Esasen ikili ağaç, değiştirilmiş bir çift bağlı listedir. Listeye göre başlıca avantajı, çabucak aranabilmesidir. Yani, eklemeler ve aramalar çok hızlı olmaktadır. Bağlı liste yapısı istediğiniz ama hızlı arama sürelerine gerek duyduğunuz uygulamalarda ikili ağaç kusursuzdur.

Elektronik tablo örneğini desteklemek amacıyla ikili ağaç kullanmak için `cell` yapısını aşağıdaki kodda gösterildiği gibi değiştirmelisiniz:

```
struct cell {
  char cell_name[9];  /* hucre adi, ornegin A1, B34 */
  char  formula[128]; /* bilgi, ornegin 10/B2 */
  struct cell *left;  /* sol alt agaci gosteren isaretci */
  struct cell *right; /* sag alt agaci gosteren isaretci */
} list entry;
```

Bölüm 22'deki `stree()` fonksiyonunu hücre adını temel alarak bir ağaç kuracak şekilde değiştirebilirsiniz. Aşağıdaki kodda, `n` parametresinin ağaca yerleştirilen yeni kaydı gösteren bir işaretçi olduğuna dikkat edin.

```
struct cell *stree(
        struct cell *root,
        struct cell *r,
        struct cell *n)
{
  if(!r) {    /* alt agactaki ilk dugum */
    n->left = NULL;
    n->right = NULL;
    if(!root) return n;  /* agactaki ilk kayit */
    if(strcmp(n->cell_name, root->cell_name) < 0)
      root->left = n;
    else
      root->right = n;
    return n;
  }
```

```
  return root;
}
```

Son olarak, `search()` fonksiyonunun değiştirilmiş versiyonunu kullanarak, adını belirttiğiniz herhangi bir hücrenin konumunu elektronik tablo üzerinde çabucak bulabilirsiniz.

```
struct cell *search_tree(
        struct cell *root,
        char *key)
{
  if(!root) return root;  /* bos agac */
  while(strcmp(root->cell_name, key)) {
    if(strcmp(root->cell_name, key) <= 0)
      root = root->right;
    else root = root->left;
    if(root == NULL) break;
  }
  return root;
}
```

İkili Ağaç Yaklaşımının Analizi

İkili ağaç, bağlı listeye göre çok daha hızlı ekleme ve arama süresiyle sonuçlanır. Hatırlarsanız, sıralı bir arama, ortalama $n/2$ karşılaştırma gerektirmektedir. Burada n, listedeki öğe sayısıdır. İkili arama buna karşın, yalnızca $\log_2 n$ karşılaştırma gerektirir (ağacın dengeli olduğu varsayımıyla). Ayrıca, ikili ağaç, bellek açısından çift bağlı liste kadar verimlidir. Ancak, bazı durumlarda ikili ağaçtan daha iyi bir alternatif mevcuttur.

Seyrek Dizilere İşaretçi Dizisi Yaklaşımı

Diyelim ki, elektronik tablonuz 26 x 100 (A1 ile Z100 arası) boyutunda ya da toplam 2.600 öğeye sahip olsun. Teorik olarak, elektronik tablo girdilerini tutmak için aşağıdaki yapı dizisini kullanabilirsiniz:

```
struct cell {
  char cell_name[9];
  char  formula[128];
} list_entry[2600];   /* 2.600 adet hucre */
```

Ancak, 2.600 x 137 (yapının ham büyüklüğü) toplam 356.200 byte büyüklüğünde belleğe karşılık gelir. Bu, tam dolu olmayan bir dizi için harcanacak fazlasıyla çok bellektir. Ne var ki, `cell` tipinde yapılan gösteren bir *işaretçi dizisi* oluşturabilirsiniz. Bu işaretçi dizisi, asıl dizilere önemli ölçüde daha az kalıcı depolama yeri gerektirecektir. Bir dizi konumuna veri atandığında, her defasında o veri için bellek ayrılacak ve işaretçi dizisinde uygun bir işaretçi o veriyi gösterecek şekilde ayarlanacaktır. Bu plan, bağlı liste ve ikili ağaç yöntemlerine nazaran üstün performans sunmaktadır. Bu tür bir işaretçi dizisi oluşturan deklarasyon şöyledir:

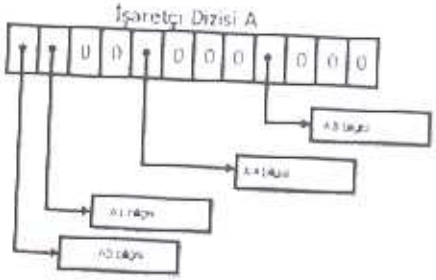
```
struct cell {
  char cell_name[9];
```

```
    char formula[128];
  } list_entry;

  struct cell *sheet[2600]; /* 2.600 işaretçiden oluşan bir dizi */
```

Bu daha küçük diziyi, elektronik tablo kullanıcısının girdiği gerçek bilgileri gösteren işaretçileri tutmak için kullanabilirsiniz. Her kayıt girildiğinde, hücre hakkındaki bilgileri gösteren bir işaretçi dizide uygun bir konumda depolanır. Şekil 23.2'de, işaretçi dizisinin seyrek dizi için destek sağlaması sayesinde, bunun bellekte nasıl görünebileceği gösterilmektedir.

İşaretçi dizisinin kullanılabilmesi için öncelikle her öğeye ilk değer olarak null atanmalıdır. Bu, o konuma hiç girdi olmadığını belirtir. Bunu gerçekleştiren fonksiyon şudur:

```
void init_sheet(void)
{
  register int t;

  for(t=0; t < 2600; ++t) sheet[t] = NULL;
}
```



ŞEKİL 23.2: Bir seyrek diziyi desteklemek üzere bir işaretçi dizisi.

Kullanıcı, bir hücre için bir formül girdiğinde, hücre konumu (adı ile tanımlanmıştır) kullanılarak **sheet** işaretçi dizisi için bir indeks üretilir. Hücre adı aşağıdaki listede gösterildiği gibi bir sayıya dönüştürülür ve indeks, hücre adından türetilir:

```
void store(struct cell *i)
{
  int loc;
  char *p;

  /* hucre adi verilince indeksi hesapla */
  loc = *(i->cell_name) - 'A'; /* sutun */
  p = &(i->cell_name[1]);
  loc += (atoi(p)-1) * 26; /* satir sayisi * satir genisligi + sutun */

  if(loc >= 2600)
    printf("Cell out of bounds.\n");
    return;

  sheet[loc] = i; /* isaretciyi diziye yerlestir */
}
```

İndeksi hesaplarken `store()`, tüm hücre adlarının büyük harf ile başladığını ve ardından bir tamsayı geldiğini varsayar; örneğin, B34, C19 gibi. Bu nedenle, `store()`'da gösterilen formül kullanıldığında A1 hücre adı 0 indeksini; B1, 1 indeksini; A2, 26 indeksini üretir. Her hücre adı benzersiz olduğu için, her indeks de benzersizdir ve her bir kaydı gösteren işaretçi uygun dizi öğesinde depolanır. Bu prosedürü bağlı liste ya da ikili ağaç versiyonlarıyla karşılaştırırsanız, bunun çok daha kısa ve basit olduğunu göreceksiniz.

`deletecell()` fonksiyonu da çok kısa olmuştur. Çıkartılacak hücrenin adı ile çağrıldığında, öğeyi gösteren işaretçiyi sıfırlar ve belleği sisteme döndürür.

```
void deletecell(struct cell *i)
{
  int loc;
  char *p;

  /* hucre adı verilince indeksi hesapla */
  loc = *(i->cell_name) - 'A'; /* sutun */
  p = &(i->cell_name[1]);
  loc += (atoi(p)-1) * 26; /* satir sayisi * satir genisligi + sutun */

  if(loc >= 2600) {
    printf("Cell out of bounds.\n");
    return;
  }
  if(!sheet[loc]) return; /* null isaretciyi serbest birakma */

  free(sheet[loc]);  /* belleği sisteme dondur */
  sheet[loc] = NULL;
}
```

Bir kez daha, bu kod bağlı liste versiyonuna göre çok daha hızlı ve basittir.

Adı verilen bir hücrenin konumu bulma süreci basittir, çünkü ad kendi başına dizi indeksini doğrudan üretmektedir. Dolayısıyla, `find()` fonksiyonu şu hali alır:

```
struct cell *find(char *cell_name)
{
  int loc;
  char *p;

  /* hucre adı verilince indeksi hesapla */
  loc = *(cell_name) - 'A'; /* sutun */
  p = &(cell_name[1]);
  loc += (atoi(p)-1) * 26; /* satir sayisi * satir genisligi + sutun */

  if(loc>=2600 || !sheet[loc]) { /* o hucreye kayit yok */
    printf("Cell not found.\n");
    return NULL;  /* bulunamadı */
  }
  else return sheet[loc];
}
```

İşaretçi Dizisi Yaklaşımının Analizi

Seyrek dizileri ele almak için işaretçi dizisi yöntemi, bağlı liste ya da ikili ağaç yöntemine kıyasla dizi öğelerine çok daha hızlı erişim sağlamaktadır. Dizi çok büyük olmadığı takdirde, işaretçi dizisi tarafından kullanılan bellek genellikle sistemin boş belleği üzerinde önemli bir harcama sayılmaz. Ancak, işaretçi dizisi kendi başına, – işaretçiler gerçek bilgiye işaret ediyor olsun ya da olmasın – her konum için bir miktar bellek kullanır. Bu belirli uygulamalar için kısıtlama olabilir, fakat genel anlamda bir problem değildir.

Hashing

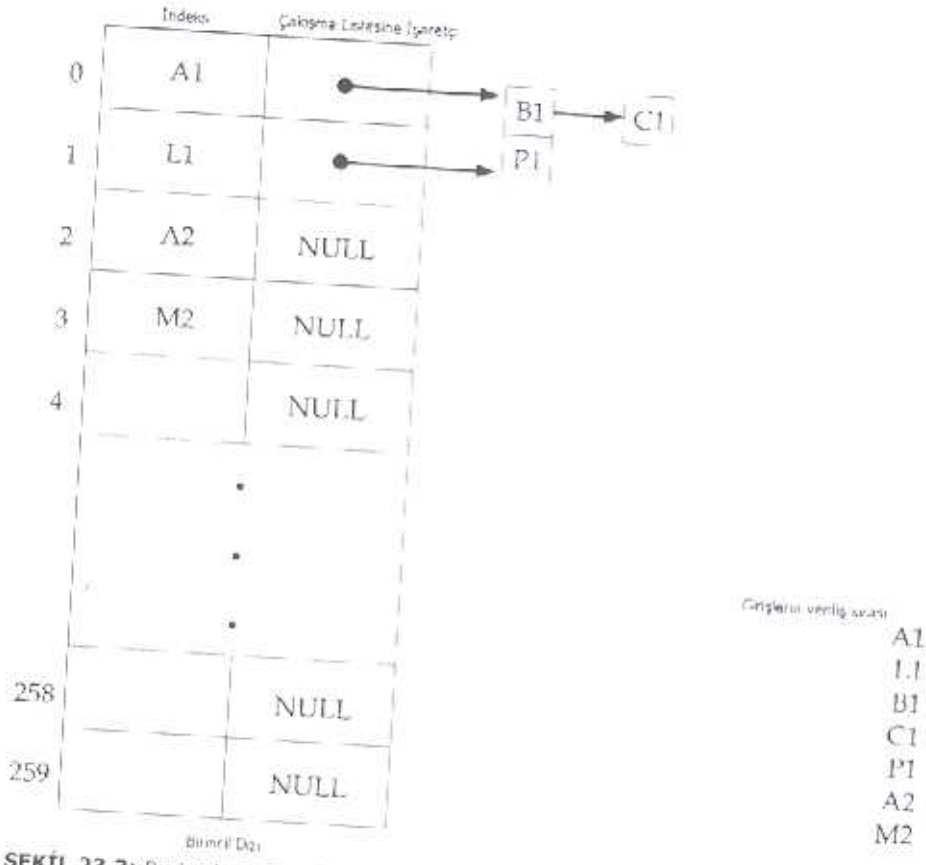
Hashing, bir dizi ögesinin indeksini doğrudan, orada depolanan bilgiden çıkartma sürecidir. Üretilen indekse *hash* denir. Alışılageldik şekliyle hashing, disk dosyalarına erişim süresini azaltma yöntemi olarak uygulanmaktadır. Ancak, aynı genel yöntemi kullanarak seyrek dizileri gerçekleyebilirsiniz. Yukarıdaki işaretçi dizisi örneğinde, *doğrudan indeksleme* denilen hashing'in özel bir biçimi kullanıldı. Bu yönteme göre, her anahtar sadece ve sadece tek bir dizi konumuyla eşlenir. Bir başka deyişle, her hash'lenen indeks benzersizdir. (İşaretçi dizisi yöntemi, doğrudan indekslenen hash gerektirmez. Bu, elektronik tablo problemi söz konusu olduğunda yalnızca belli bir yöntemdi.) Gerçek uygulamalarda bu tür doğrudan hashing yaklaşımları çok azdır; üstelik daha esnek yöntemler gereklidir. Bu bölümde, daha fazla güç ve esnekliğe olanak sağlamak için hashing'in nasıl genelleştirilebileceği gösterilmektedir.

Elektronik tablo örneği, en ıtız ortamlarda bile çalışma sayfasındaki her hücrenin kullanılmayacağını açıkça ortaya koymaktadır. Hemen hemen her durumda, gerçek kayıtların potansiyel konumların yüzde 10'undan fazlasını kaplamayacağını varsayalım. Dolayısıyla, elektronik tablo eğer 26 x 100 (2.600 konum) boyutuna sahipse, bir defada hücrelerin yalnızca yaklaşık 260'ı gerçekten kullanılmaktadır. Bundan çıkarılacak sonuç şudur: Tüm kayıtları tutmak için gerekli en büyük dizi normalde yalnızca 260 öğeden oluşacaktır. Fakat, mantıksal dizi konumları bu nispeten küçük fiziksel diziye nasıl eşlenecek ve ondan nasıl erişilecektir? Ayrıca, bu dizi dolarsa ne olacaktır? Aşağıdaki anlatımda bir olası çözüm açıklanmaktadır.

Bir hücrenin verisi, elektronik tablo (bu, mantıksal dizidir) kullanıcısı tarafından girildiğinde, hücre adı ile tanımlanan hücre konumu kullanılarak daha küçük fiziksel dizi için bir indeks (bir hash) üretilir. Hashing'le bağlantılı olarak fiziksel diziye ayrıca *birincil dizi* de denir. İndeks, hücre adından türetilir. Hücre adı ise işaretçi dizisi örneğinde olduğu gibi bir sayıya dönüştürülmüştür. Ancak, bu sayı daha sonra 10'a bölünerek birincil diziye ilk giriş noktası oluşturulmuştur. (Hatırlarsanız, bu örnekte fiziksel dizinin büyüklüğü mantıksal dizinin yalnızca yüzde 10'u kadardır.) Bu indeksin gösterdiği konum boş ise, mantıksal indeks ve ilgili değer oraya depolanır. Ancak, 10 mantıksal konum aslında tek bir fiziksel konuma eşlendiği için, hash çakışmaları meydana gelebilir. Bu durumu söz konusu olunca, veri girişini tutmak için bir bağlı liste – kimi zaman buna *çakışma listesi* de denir – kullanılır. Birincil dizideki her kayıt, ayrı bir çakışma listesi ile ilişkilendirilir. Kuşkusuz, bu listeler bir çakışma olana kadar sıfır uzunluğundadır. Bu durum Şekil 23.3'te gösterilmiştir.

Diyelim ki, mantıksal dizi indeksi verilen bir öğeyi fiziksel dizide bulmak istiyorsunuz. Öncelikle, mantıksal indeksi karşılık gelen hash değerine çevirin ve hash tarafından üretilen indeksteki fiziksel diziyi kontrol ederek, orada depolanan mantıksal indeksin aradığınız indeksle eşle-

nip eşlenmediğini görün. Eşleniyorsa, bilgiyi döndürün. Aksi halde, uygun indeksi bulana kadar ya da zincirin sonuna ulaşana dek çakışma listesini takip edin.



ŞEKİL 23.3: Bir hashing örneği.

`primary` denilen bir yapı dizisi kullanan hashing örneği aşağıda gösterilmiştir:

```
#define MAX 260

struct htype {
  int index;    /* mantıksal indeks */
  int val;      /* dizi ögesinin gerçek değeri */
  struct htype *next; /* aynı hashe sahip bir sonraki değeri gösteren işaretçi */
} primary[MAX];
```

Bu dizinin kullanılabilmesi için dizi öncelikle ilk kullanıma hazırlanmalıdır. Aşağıdaki fonksiyon, `index` alanına, boş öğeye işaret etmek için, ilk değer olarak -1 (tanım gereği, üretilemeyen bir değer) atar. `next` alanındaki `NULL`, boş hash zincirine işaret eder.

```
/* hash dizisini ilk kullanime hazirla. */
void init(void)
{
  register int i;

  for (i=0; i<MAX; i++) {
    primary[i].index = -1;
    primary[i].next = NULL;  /* null zinciri */
    primary[i].val = 0;
  }
}
```

`store()` prosedürü, bir hücre adını birincil dizi için hash'lenmiş bir indekse dönüştürür. Hash'lenmiş değer tarafından doğrudan gösterilen konum dolu ise, prosedür bir önceki bölümde geliştirilen `slstore()`'un değiştirilmiş bir versiyonunu kullanarak bu kaydı otomatik olarak çakışma listesine ekler. Mantıksal indeks depolanmalıdır, çünkü o öğe tekrar erişildiğinde bu indekse gerek olacaktır. Bu fonksiyonlar aşağıda gösterilmiştir:

```
/* hash'i hesaplar ve degeri depolar. */
void store(char *cell_name, int v)
{
  int h, loc;
  struct htype *p;

  /* hash degerini uret */
  loc = *cell_name - 'A'; /* sutun */
  loc += (atoi(&cell_name[1])-1) * 26; /* satir * genislik + sutun */
  h = loc%10; /* hash */

  /* Dolu degilse o konumda depola ya da
     mantiksal indeksler uyusuyorsa orada depola - yani, guncelle.
  */
  if(primary[h].index==-1 || primary[h].index==loc) {
    primary[h].index = loc;
    primary[h].val = v;
    return;
  }

  /* aksi halde, olustur ya da cakisma listesine ekle */
  p = (struct htype *) malloc(sizeof(struct htype));
  if(!p) {
    printf("Out of Memory");
    return;
  }
  p->index = loc;
  p->val = v;
  slstore(p, &primary[h]);
}

/* Ogeleri cakisma listesine ekle. */
void slstore(struct htype *i,
             struct htype *start)
```

```
{
  struct htype *old, *p;

  old = start;
  /* listenin sonunu bul */
  while(start) {
    old = start;
    start = start->next;
  }
  /* yeni kaydı bagla */
  old->next = i;
  i->next = NULL;
}
```

Bir öğenin değerini bulmadan önce, programınız önce hash'i hesaplamalıdır, sonra fiziksel dizide depolanan mantıksal indeksin talep edilen mantıksal dizinin indeksi ile eşleniş eşlenmediğini kontrol etmelidir. Eşleniyorsa, o değer döndürülür; aksi halde, çakışma zincini araştırılır. Bu görevleri gerçekleştiren `find()` fonksiyonu burada gösterilmiştir:

```
/* hash'i hesaplar ve degeri dondurur. */
int find(char *cell_name)
{
  int h, loc;
  struct htype *p;

  /* hash degerini uret */
  loc = *cell_name - 'A'; /* sutun */
  loc += (atoi(&cell_name[1])-1) * 26; /* satir * genislik + sutun */
  h = loc/10;

  /* bulunduysa degeri dondur */
  if(primary[h].index == loc) return(primary[h].val);
  else { /* cakisma listesine bak */
    p = primary[h].next;
    while(p) {
      if(p->index == loc) return p->val;
      p = p->next;
    }
    printf("Not in Array\n");
    return -1;
  }
}
```

Silme fonksiyonunu geliştirmek size alıştırma olarak bırakılmıştır. (*İpucu*: Ekleme sürecini tam tersine çevirmeniz yeterlidir.)

Yukarıdaki hash algoritmasının çok basit olduğunu aklınızdan çıkartmayın. Genelde, birincil dizideki indekslerin daha düzgün dağılımlarını sağlamak için daha karmaşık bir yöntem kullanırsınız. Böylece, uzun hash zincirlerini önlersiniz. Ancak, temel prensip aynıdır.

Hashing'in Analizi

En iyi durumda (bu oldukça enderdir), hash tarafından oluşturulan fiziksel indekslerin her biri benzersizdir ve erişim zamanı "doğrudan indekslemeye" yakındır. Bu, hiç çakışma listesi oluşturulmayacak demektir. Ayrıca, tüm aramalar esasen doğrudan erişimlidir. Ancak, bu nadiren söz konusu olacaktır, çünkü bu durum mantıksal indekslerin fiziksel indeks uzayı içinde düzgün dağılmasını gerektirmektedir. En kötü durumda (bu da enderdir), hash'lenmiş bir şema, bağlı listeye indirgenir. Bu, mantıksal indekslerin hash'lenmiş değerlerinin tümü aynı olduğunda söz konusudur. Ortalama durumda (en olası olan budur), hash yöntemi, bir doğrudan indeks kullanarak harcanan zamana ek olarak hash zincirlerinin ortalama uzunluğuyla orantılı bir sabit süre içinde spesifik bir öğeye erişebilir. Bir seyrek diziyi desteklemek amacıyla hashing kullanırken en kritik faktör, uzun çarpışma listelerinden kaçınmak için fiziksel indeksi düzgün olarak dağıtan bir hashing algoritmasının kullanıldığından emin olmaktır.

Bir Yaklaşım Tercih Etmek

Bir seyrek diziyi uygulamak için bağlı liste, ikili ağaç, işaretçi dizisi ya da hashing yaklaşımlarından birini kullanmaya karar verirken hızı ve belleğin verimli kullanılmasını göz önünde bulundurmalısınız. Ayrıca, seyrek dizinizin muhtemelen az mı, yoksa çok mu öğe içereceğini de dikkate almalısınız.

Mantıksal dizi çok seyrek olduğunda, bellek açısından en verimli yaklaşımlar bağlı listeler ve ikili ağaçlardır. Çünkü, yalnızca gerçekten kullanılan dizi öğeleri kendilerine ayrılmış belleğe sahiptir. Bağlantılar tek başına çok az ek bellek gerektirirler ve genellikle bu ihmal edilebilir elkiye sahiptir. İşaretçi dizisi tasarımı, bazı öğeleri kullanılmıyor olsa bile işaretçi dizisinin bütün olarak mevcut olmasını gerektirir. İşaretçi dizisinin bütünüyle bellekte olmasının yanı sıra, ayrıca uygulamanın kullanması için yeterli bellek de kalmalıdır. Bu, başkaları için hiçbir sorun teşkil etmeyebilirken, belirli uygulamalar için ciddi problem doğurabilir. Yaklaşık boş bellek miktarını genellikle hesaplayabilirsiniz ve onun programınız için yeterli olup olmadığını belirleyebilirsiniz. Hashing yöntemi, işaretçi dizisi ile bağlı liste/ikili ağaç yaklaşımları arasında bir yerdedir. Gerçi hashing yöntemi de, tümüyle kullanılmıyor olsa bile birincil dizinin tümüyle var olmasını gerektirmesine karşın, yine de işaretçi dizisinden daha küçük olacaktır.

Mantıksal dizi oldukça dolu olduğunda durum çok değişir. Bu durumda, işaretçi dizisi ile hashing çok daha cazip olur. Ayrıca, mantıksal dizi ne kadar dolu olursa olsun, işaretçi dizisinde bir öğeyi bulmak için geçen zaman sabittir. Hashing yaklaşımı için arama süresi sabit olmamakla birlikte, düşük bir değer ile sınırlıdır. Ancak, dizinin öğe sayısı giderek arttıkça bağlı liste ve ikili ağaç için ortalama arama süresi de artar. Tutarlı erişim süreleri önemli olduğunda bunu aklınızdan çıkartmak istemeyeceksiniz.