Chapter 9

TABLES AND INFORMATION RETRIEVAL

1. Introduction: Breaking the lg n Barrier
2. Rectangular Arrays
3. Tables of Various Shapes
4. Tables: A New Abstract Data Type
5. Application: Radix Sort
6. Hashing
7. Analysis of Hashing
8. Conclusions: Comparison of Methods
9. Application: The Life Game Revisited

Introduction: Breaking the lg n Barrier

- By use of key comparisons alone, it is impossible to complete a search of n items in fewer than $\lg n$ comparisons, on average (lower*bound_search_thm*).
- Ordinary table lookup or array access requires only constant time $O(1)$.
- Both table lookup and searching share the same essential purpose, that of *information retrieval*. The *key* used for searching and the *index* used for table lookup have the same essential purpose: one piece of information that is used to locate further information.
- Both table lookup and searching algorithms provide *functions* from a set of keys or indices to locations in a list or array.
- In this chapter we study ways to implement and access various kinds of tables in contiguous storage.
- Several steps may be needed to retrieve an entry from some kinds of tables, but the time required remains $O(1)$. It is bounded by a constant that does not depend on the size of the table. Thus table lookup can be more efficient than any searching method.
- We shall implement abstractly defined tables with arrays. In order to distinguish between the abstract concept and its implementation, we introduce:

Convention

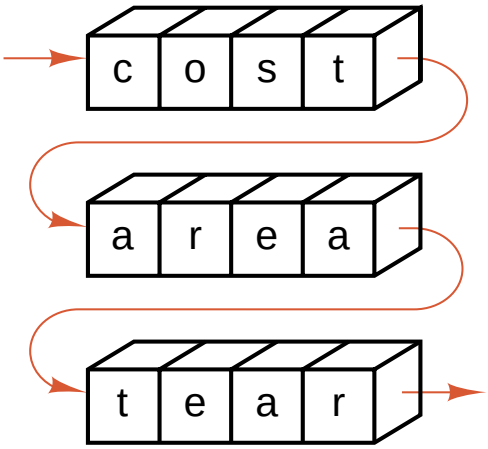
The index defining an entry of an abstractly defined table
is enclosed in parentheses,
whereas the index of an entry of an array
is enclosed in square brackets.

Rectangular Arrays

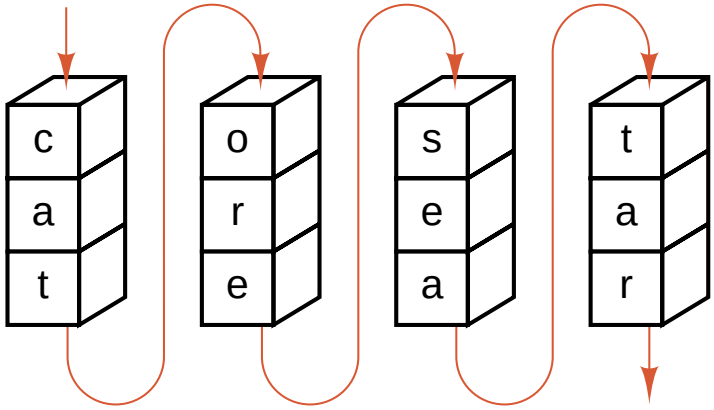
Rectangular table

c	o	s	t
a	r	e	a
t	e	a	r

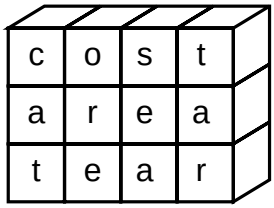
Row-major ordering:



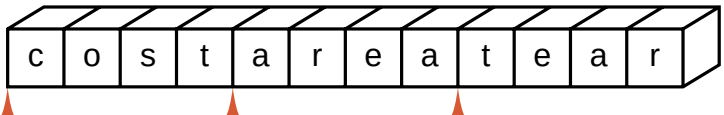
Column-major ordering:



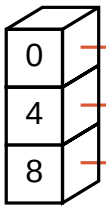
In row-major ordering, entry $[i, j]$ goes to position $ni + j$.



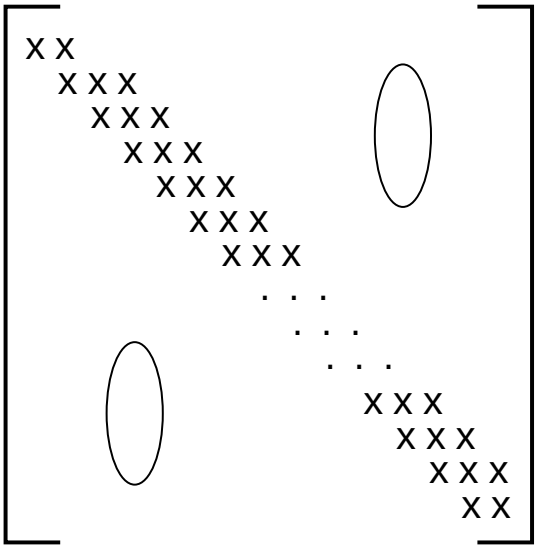
is represented in row-major order as



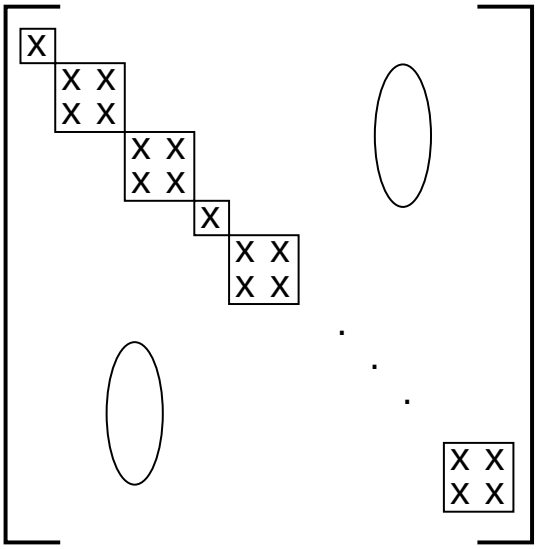
Access array



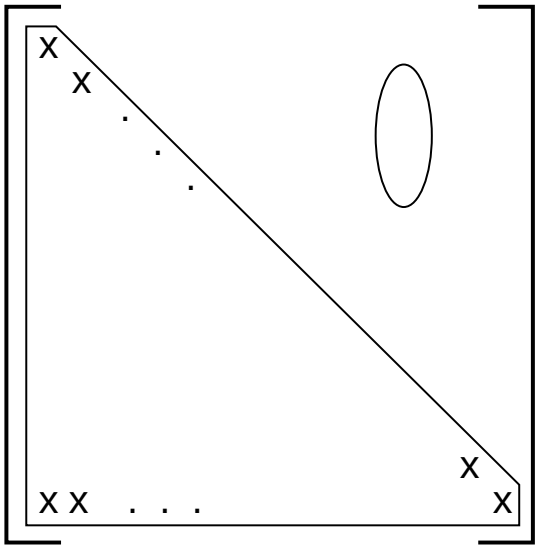
Matrices of Various Shapes



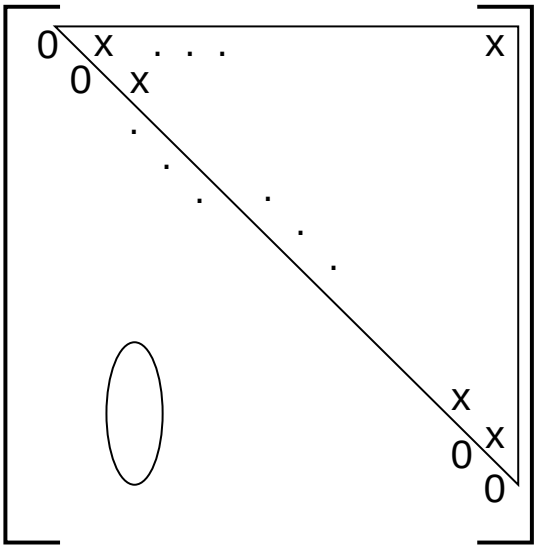
Tri-diagonal matrix



Block diagonal matrix

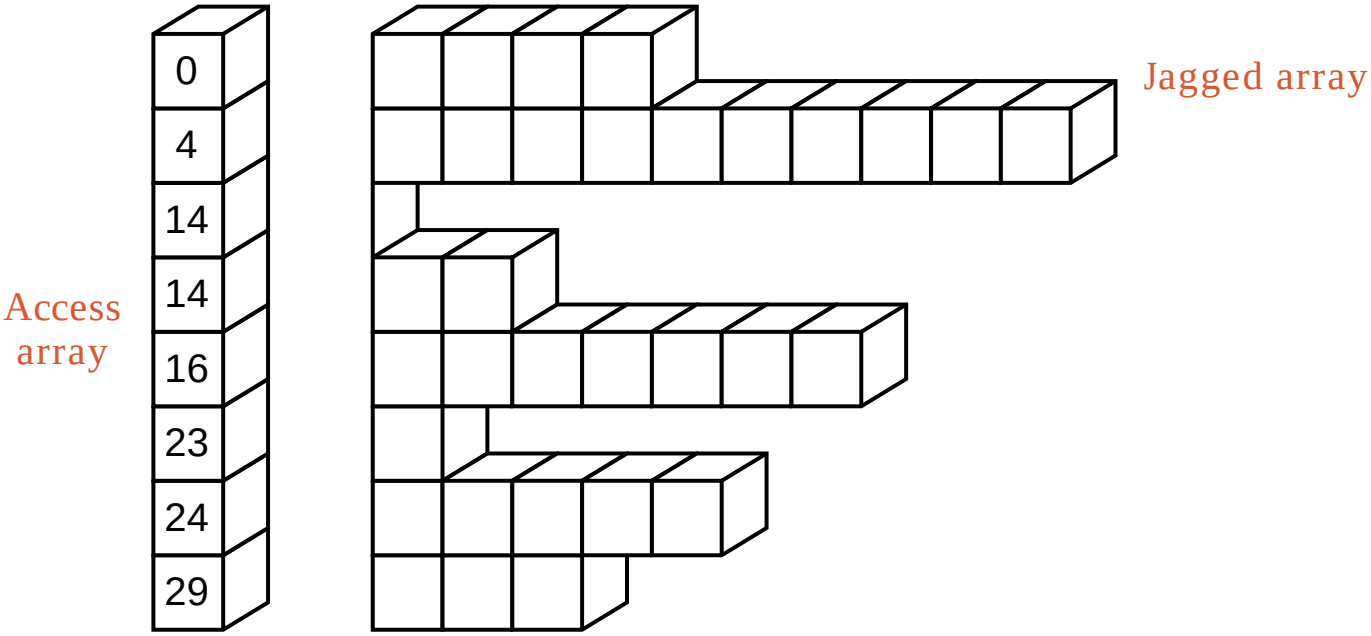


Lower triangular matrix

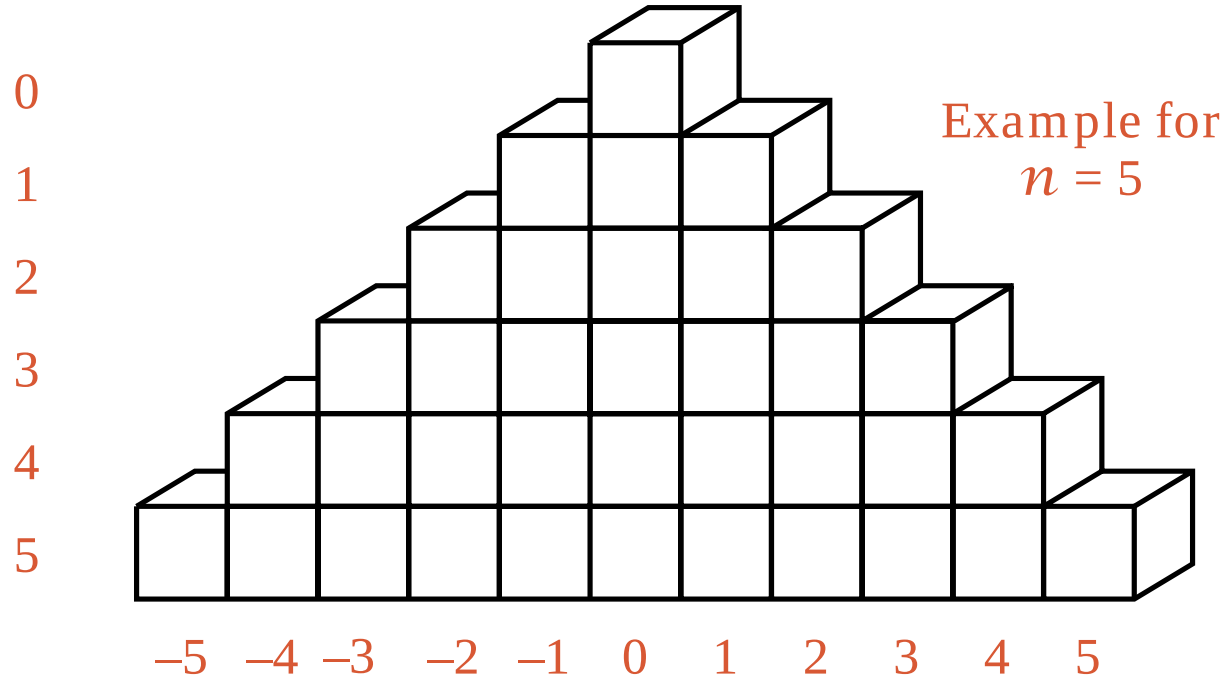


Strictly upper triangular matrix

Jagged Table with Access Table



Symmetrically Triangular Table



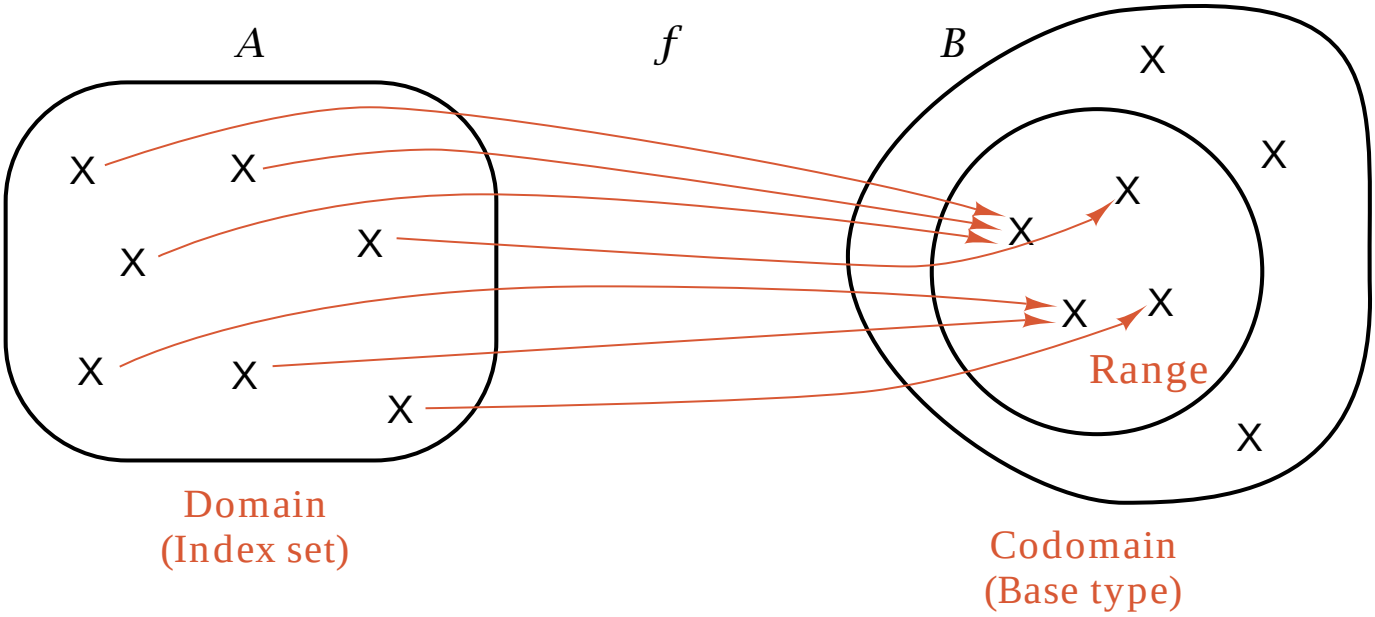
Inverted Table

Index	*Name*	*Address*	*Phone*
1	Hill, Thomas M.	High Towers #317	2829478
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 King Street	2495723
6	Byers, Carolyn	118 Maple Street	4394231
7	Moody, C. L.	High Towers #210	2822214

Access Tables

Name	*Address*	*Phone*
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

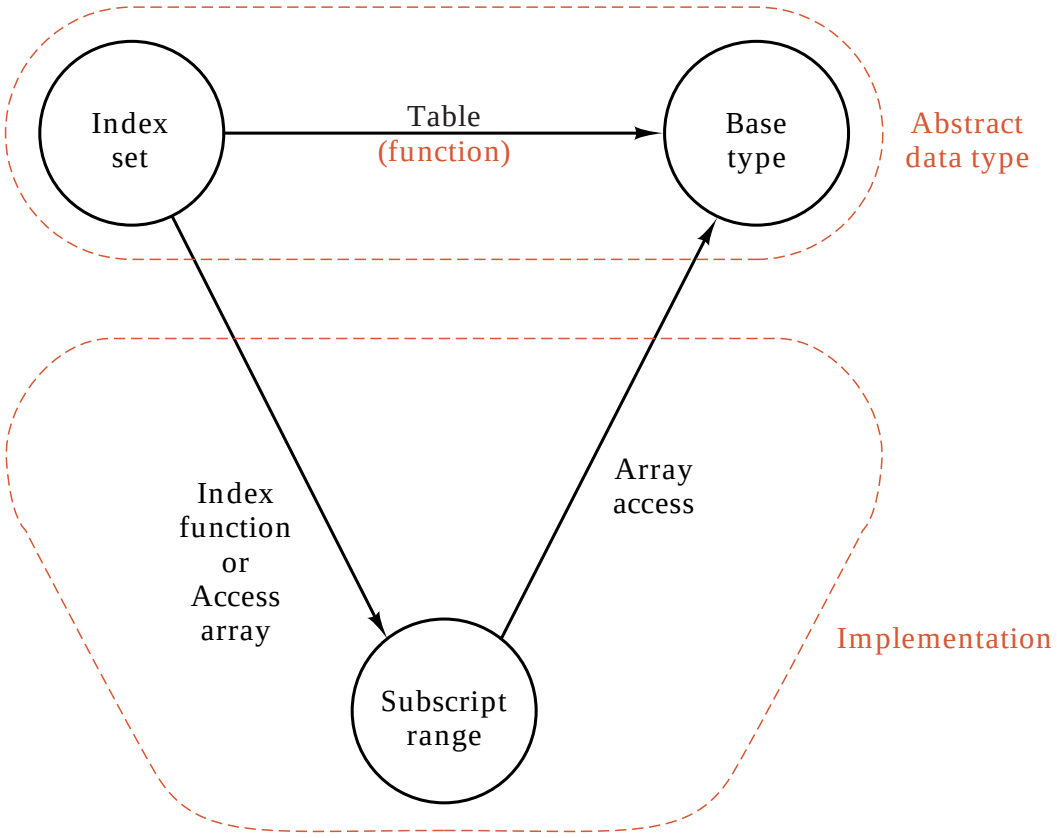
Functions



- In mathematics a ***function*** is defined in terms of two sets and a correspondence from elements of the first set to elements of the second. If f is a function from a set A to a set B, then f assigns to each element of A a unique element of B.
- The set A is called the ***domain*** of f, and the set B is called the ***codomain*** of f.
- The subset of B containing just those elements that occur as values of f is called the ***range*** of f.

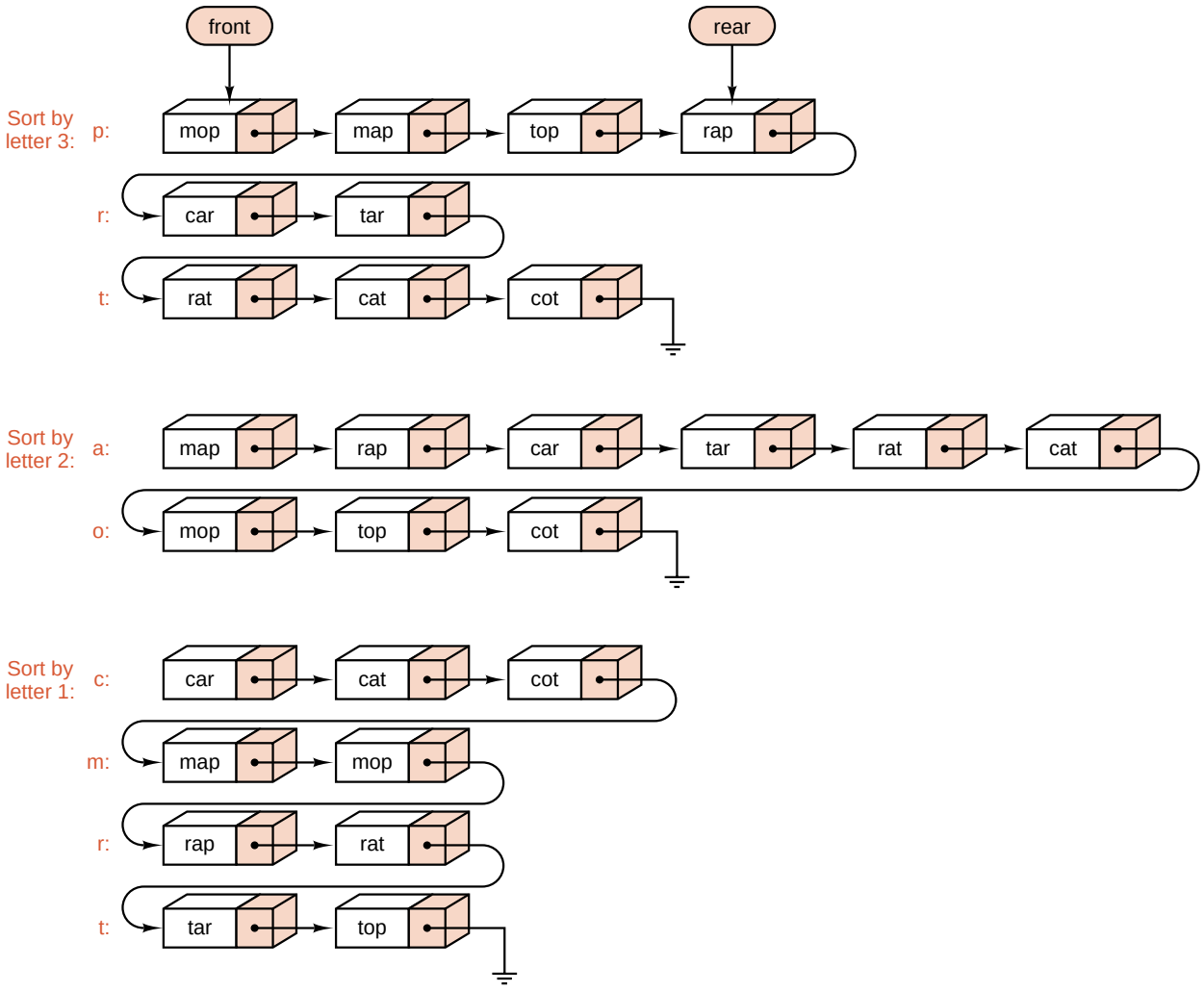
DEFINITION A ***table*** with index set I and base type T is a function from I to T together with the following operations.

1. *Table access*: Evaluate the function at any index in I.
2. *Table assignment*: Modify the function by changing its value at a specified index in I to the new value specified in the assignment.
3. *Creation*: Set up a new function.
4. *Clearing*: Remove all elements from the index set I, so there is no remaining domain.
5. *Insertion*: Adjoin a new element x to the index set I and define a corresponding value of the function at x.
6. *Deletion*: Delete an element x from the index set I and restrict the function to the resulting smaller domain.



Radix Sort

Initial order	Sorted by letter 3	Sorted by letter 2	Sorted by letter 1
rat	mop	map	car
mop	map	rap	cat
cat	top	car	cot
map	rap	tar	map
car	car	rat	mop
top	tar	cat	rap
cot	rat	mop	rat
tar	cat	top	tar
rap	cot	cot	top



Linked Implementation of Radix Sort

List definition:

```
template <class Record>
class Sortable_list:  public List<Record> {
public:                              //    sorting methods
   void radix_sort( );
   //    Specify any other sorting methods here.
private:                             //    auxiliary functions
   void rethread(Queue queues[ ]);
};
```

Record definition:

```
class Record {
public:
   char key_letter(int position) const;
   Record( );                        //    default constructor
   operator Key( ) const;            //    cast to Key
//    Add other methods and data members for the class.
};
```

Sorting Method, Linked Radix Sort

```
const int max_chars = 28;
template <class Record>
void Sortable_list<Record> :: radix_sort( )
/* Post:  The entries of the Sortable_list have been sorted so all their keys are in
          alphabetical order.
   Uses:  Methods from classes List, Queue, and Record;
          functions position and rethread.  */


{
   Record data;
   Queue queues[max_chars];
   for (int position = key_size – 1; position >= 0; position––) {
      //    Loop from the least to the most significant position.
      while (remove(0, data) ==  success) {
         int queue_number = alphabetic_order(data.key_letter(position));
         queues[queue_number].append(data);   //    Queue operation.
      }
      rethread(queues);                       //    Reassemble the list.
   }
}
```

Auxiliary Functions, Linked Radix Sort

Selecting a queue:

```
int alphabetic_order(char c)
/* Post: The function returns the alphabetic position of character c, or it returns 0
         if the character is blank. */
{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c - 'a' + 1;
   if ('A' <= c && c <= 'Z') return c - 'A' + 1;
   return 27;
}
```

Connecting the queues:

```
template <class Record>
void Sortable_list<Record> :: rethread(Queue queues[ ])
/* Post: All the queues are combined back to the Sortable_list, leaving all the
         queues empty.
   Uses: Methods of classes List, and Queue. */
{
   Record data;
   for (int i = 0; i < max_chars; i++)
      while (!queues[i].empty( )) {
         queues[i].retrieve(data);
         insert(size( ), data);
         queues[i].serve( );
      }
}
```

Analysis of Radix Sort

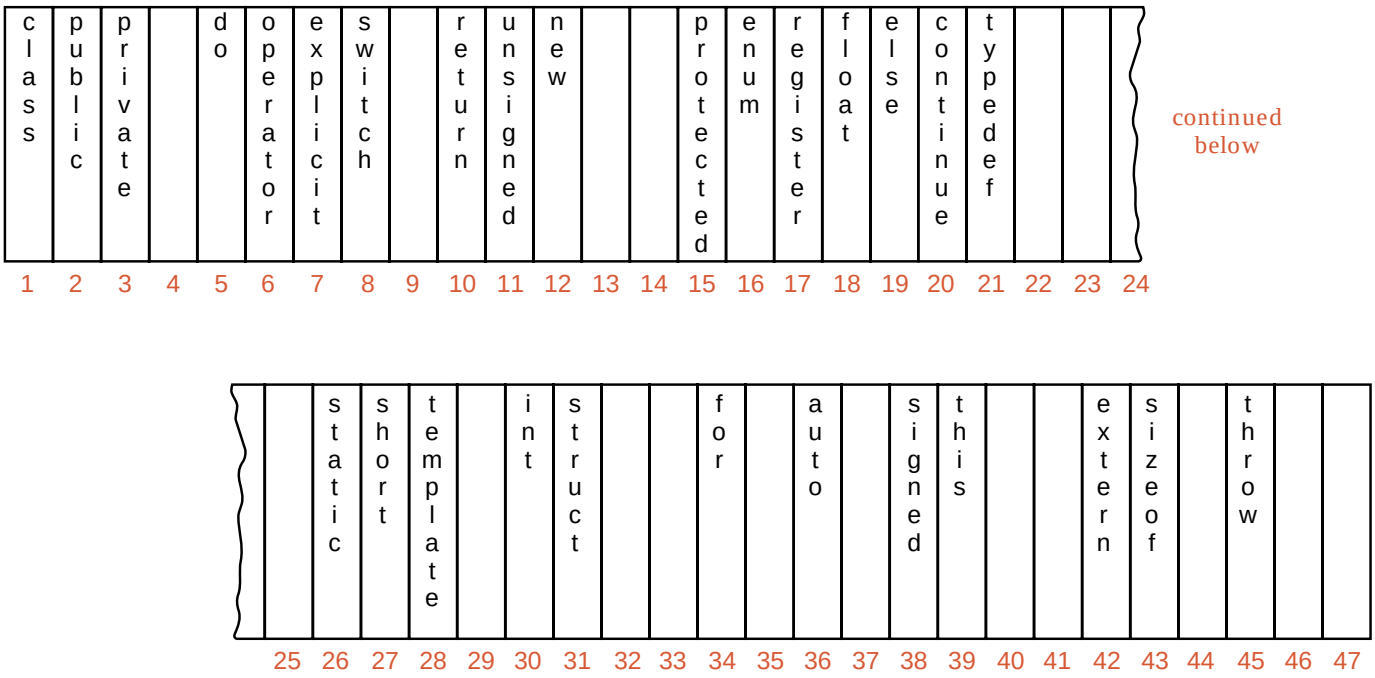
- The time used by radix sort is $\Theta(nk)$, where n is the number of items being sorted and k is the number of characters in a key.

- The relative performance of radix sort to other methods will relate to the relative sizes of nk and $n \lg n$; that is, of k and $\lg n$.

- If the keys are long but there are relatively few of them, then k is large and $\lg n$ relatively small, and other methods (such as mergesort) will outperform radix sort.

- If k is small (the keys are short) and there are a large number of keys, then radix sort will be faster than any other method we have studied.

Hash Tables



- Start with an *array* that holds the hash table.
- Use a *hash function* to take a key and map it to some index in the array. This function will generally map several different keys to the same index.
- If the desired record is in the location given by the index, then we are finished; otherwise we must use some method to resolve the *collision* that may have occurred between two records wanting to go to the same location.
- To use hashing we must (a) find good hash functions and (b) determine how to resolve collisions.

Choosing a Hash Function

- A hash function should be easy and quick to compute.
- A hash function should achieve an even distribution of the keys that actually occur across the range of indices.
- The usual way to make a hash function ia to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that will be uniformly distributed over the range of indices.
- Note that there is nothing random about a hash function. If the function is evaluated more than once on the same key, then it must give the same result every time, so the key can be retrieved without fail.
- Truncation: Sometimes we ignore part of the key, and use the remaining part as the index.
- Folding: We may partition the key into several parts and combine the parts in a convenient way.
- Modular arithmetic: We may convert the key to an integer, divide by the size of the index range, and take the remainder as the result.
- A better spread of keys is often obtained by taking the size of the table (the index range) to be a prime number.

C++ Example of a Hash Function

We write a hash function in C++ for transforming a key consisting of eight alphanumeric characters into an integer in the range

$$0 \,..\, \text{hash\_size} - 1.$$

We start with a class Key with the methods and functions of the following definition:

```
class Key:  public String{
public:
    char key_letter(int position) const;
    void make_blank( );
    //     Add constructors and other methods.
};
```

We inherit the methods of the **class** String from Chapter 6. The method key_letter(**int** position) must return the character in a particular position of the Key, or return a blank if the Key has length less than n. The final method make_blank sets up an empty Key.

```
int hash(const Key &target)
/* Post:  target has been hashed, returning a value between 0 and hash_size − 1.
   Uses:  Methods for the class Key.  */
{
   int value = 0;
   for (int position = 0;  position < 8;  position++)
      value = 4 * value + target.key_letter(position);
   return value % hash_size;
}
```

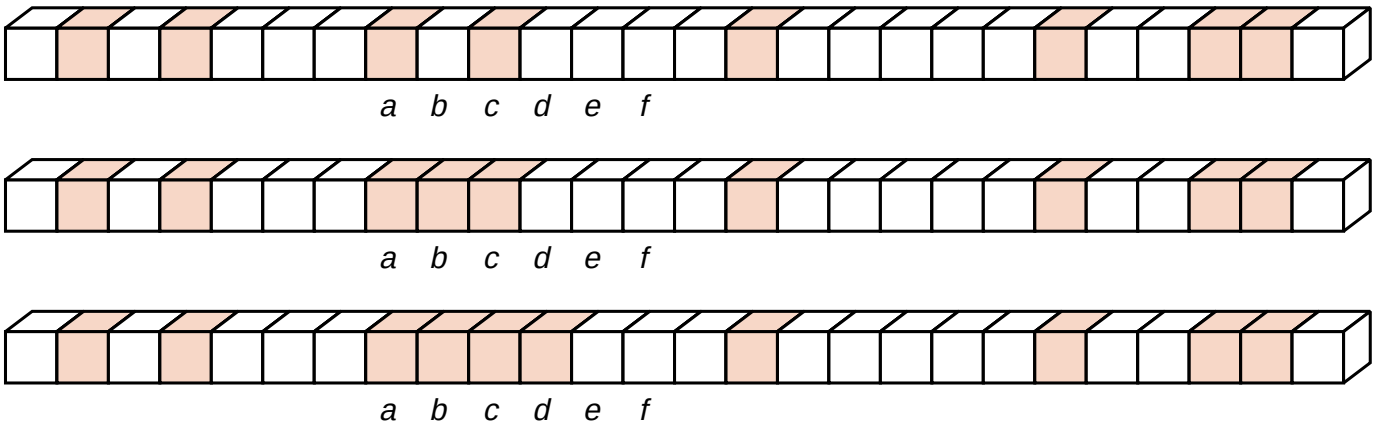
We have simply added the integer codes corresponding to each of the eight characters, multiplying by 4 each time.

Collision Resolution with Open Addressing

Linear Probing:

Linear probing starts with the hash address and searches sequentially for the target key or an empty position. The array should be considered circular, so that when the last location is reached, the search proceeds to the first location of the array.

Clustering:



Quadratic Probing:

If there is a collision at hash address h, *quadratic probing* goes to locations $h + 1$, $h + 4$, $h + 9$, . . . , that is, at locations $h + i^2$ (mod hashsize) for $i = 1, 2, \ldots$.

Other methods:

- Key-dependent increments;
- Random probing.

Hash Table Specifications

```
const int hash_size = 997;          //  a prime number of appropriate size
class Hash_table {
public:
   Hash_table( );
   void clear( );
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

```
Hash_table :: Hash_table( );
```

Post: The hash table has been created and initialized to be empty.

```
void Hash_table :: clear( );
```

Post: The hash table has been cleared and is empty.

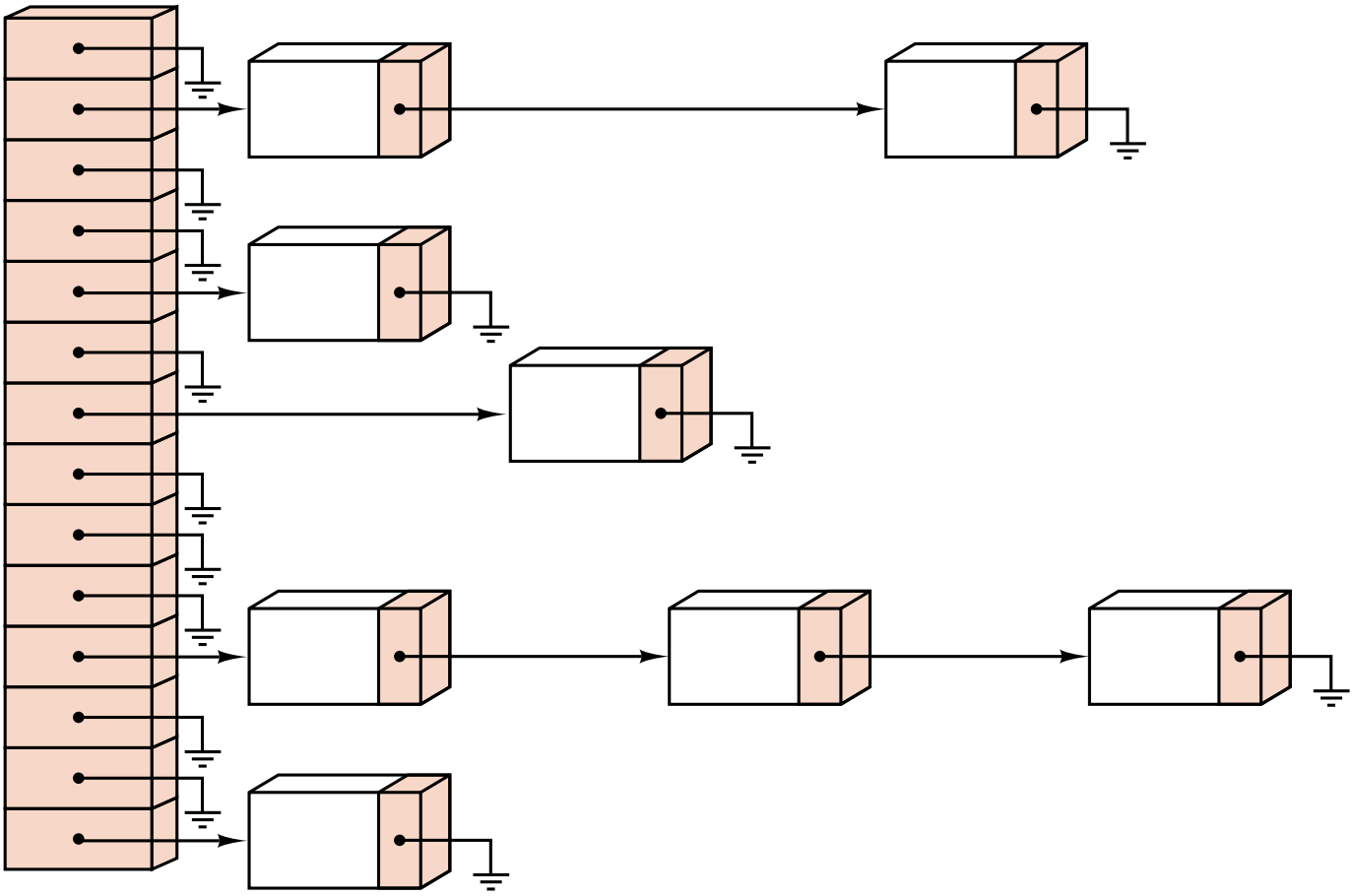
```
Error_code Hash_table :: retrieve(const Key &target,
                                  Record &found) const;
```

Post: If an entry in the hash table has key equal to target, then found takes on the value of such an entry, and success is returned. Otherwise, not_present is returned.

Hash Table Insertion

```
Error_code Hash_table :: insert(const Record &new_entry)
/* Post:  If the Hash_table is full, a code of overflow is returned.  If the table already
          contains an item with the key of new_entry a code of duplicate_error is re-
          turned.  Otherwise:  The Record new_entry is inserted into the Hash_table
          and success is returned.
   Uses:  Methods for classes Key, and Record.  The function hash.  */
{
   Error_code result = success;
   int probe_count,                  //  Counter to be sure that table is not full.
       increment,                    //  Increment used for quadratic probing.
       probe;                        //  Position currently probed in the hash table.
   Key null;                         //  Null key for comparison purposes.
   null.make_blank( );

   probe = hash(new_entry);
   probe_count = 0;
   increment = 1;

   while (table[probe] != null       //  Is the location empty?
          && table[probe] != new_entry  //  Duplicate key?
          && probe_count < (hash_size + 1)/2) {  //  Has overflow occurred?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;                //  Prepare increment for next iteration.
   }
   if (table[probe] == null) table[probe] = new_entry;
                                     //  Insert new entry.
   else if (table[probe] == new_entry) result = duplicate_error;
   else result = overflow;           //  The table is full.
   return result;
}
```

Chained Hash Tables



- The linked lists from the hash table are called ***chains***.
- If the records are large, a chained hash table can save space.
- Collision resolution with chaining is simple; clustering is no problem.
- The hash table itself can be smaller than the number of records; overflow is no problem.
- Deletion is quick and easy in a chained hash table.
- If the records are very small and the table nearly full, chaining may take more space.

Code for Chained Hash Tables

Definition:

```
class Hash_table {
public:
  //    Specify methods here.
private:
  List<Record> table[hash_size];
};
```

The **class** List can be any one of the generic linked implementations of a list studied in Chapter 6.

Constructor:

The implementation of the constructor simply calls the constructor for each list in the array.

Clear:

To clear the table, we must clear the linked list in each of the table positions, using the List method clear( ).

Retrieval:

```
sequential_search(table[hash(target)], target, position);
```

Insertion:

```
table[hash(new_entry)].insert(0, new_entry);
```

Deletion:

```
Error_code Hash_table::remove(const Key_type &target,
                              Record &x);
```

Post: If the table has an entry with key equal to target, a code of success is returned, the entry is deleted from the hash table and recorded in x. Otherwise a code of not_present is returned.

The Birthday Surprise

How many randomly chosen people need to be in a room before it becomes likely that two people will have the same birthday (month and day)?

The probability that m people all have different birthdays is

$$\frac{364}{365} * \frac{363}{365} * \frac{362}{365} * \cdots * \frac{365 - m + 1}{365}.$$

This expression becomes less than 0.5 whenever $m \geq 23$.

For hashing, the birthday surprise says that for any problem of reasonable size, collisions will almost certainly occur.

Analysis of Hashing

A ***probe*** is one comparison of a key with the target.

The ***load factor*** of the table is $\lambda = n/t$, where n positions are occupied out of a total of t positions in the table.

Retrieval from a chained hash table with load factor λ requires approximately $1 + \frac{1}{2}\lambda$ probes in the successful case and λ probes in the unsuccessful case.

Retrieval from a hash table with open addressing, random probing, and load factor λ requires approximately

$$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

probes in the successful case and $1/(1-\lambda)$ probes in the unsuccessful case.

Retrieval from a hash table with open addressing, linear probing, and load factor λ requires approximately

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

probes in the successful case and in the unsuccessful case, respectively.

Theoretical comparisons:

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random probes	1.05	1.4	2.0	2.6	4.6	—
Open, Linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search, expected number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random probes	1.1	2.0	5.0	10.0	100.	—
Open, Linear probes	1.12	2.5	13.	50.	5000.	—

Empirical comparisons:

Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search, average number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Open, Linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search, average number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Quadratic probes	1.13	2.2	5.2	11.9	126.	—
Open, Linear probes	1.13	2.7	15.4	59.8	430.	—

Conclusions: Comparison of Methods

We have studied four principal methods of information retrieval, the first two for lists and the second two for tables. Often we can choose either lists or tables for our data structures.

- Sequential search is $\Theta(n)$.

  Sequential search is the most flexible method. The data may be stored in any order, with either contiguous or linked representation.

- Binary search is $\Theta(\log n)$.

  Binary search demands more, but is faster: The keys must be in order, and the data must be in random-access representation (contiguous storage).

- Table lookup is $\Theta(1)$.

  Ordinary lookup in contiguous tables is best, both in speed and convenience, unless a list is preferred, or the set of keys is sparse, or insertions or deletions are frequent.

- Hash-table retrieval is $\Theta(1)$.

  Hashing requires the most structure, a peculiar ordering of the keys well suited to retrieval from the hash table, but generally useless for any other purpose. If the data are to be available for human inspection, then some kind of order is needed, and a hash table is inappropriate.

Application: The Life Game Revisited

- The Life cells are supposed to be on an unbounded grid, not a finite array as used in Chapter 1.
- In the class Life, we would like to have the declaration

```
class Life {
public:
   //    methods
private:
   bool map[int][int];
   //    other data and auxiliary functions
};
```

  which is, of course, illegal.
- Use a *hash table* to represent the data member map (sparse array).
- The main function remains unchanged from Chapter 1.
- Rewrite the method update so that it uses a hash table to look up the status of cells.
- For any given cell in a configuration, we can determine the number of living neighbors by looking up the status of each neighboring cell.
- Candidates that might live in the coming generation are those that are alive and their (dead) neighbors. Method update will traverse these cells, determine their neighbor counts by using the hash table, and select those cells that will live in the next generation.

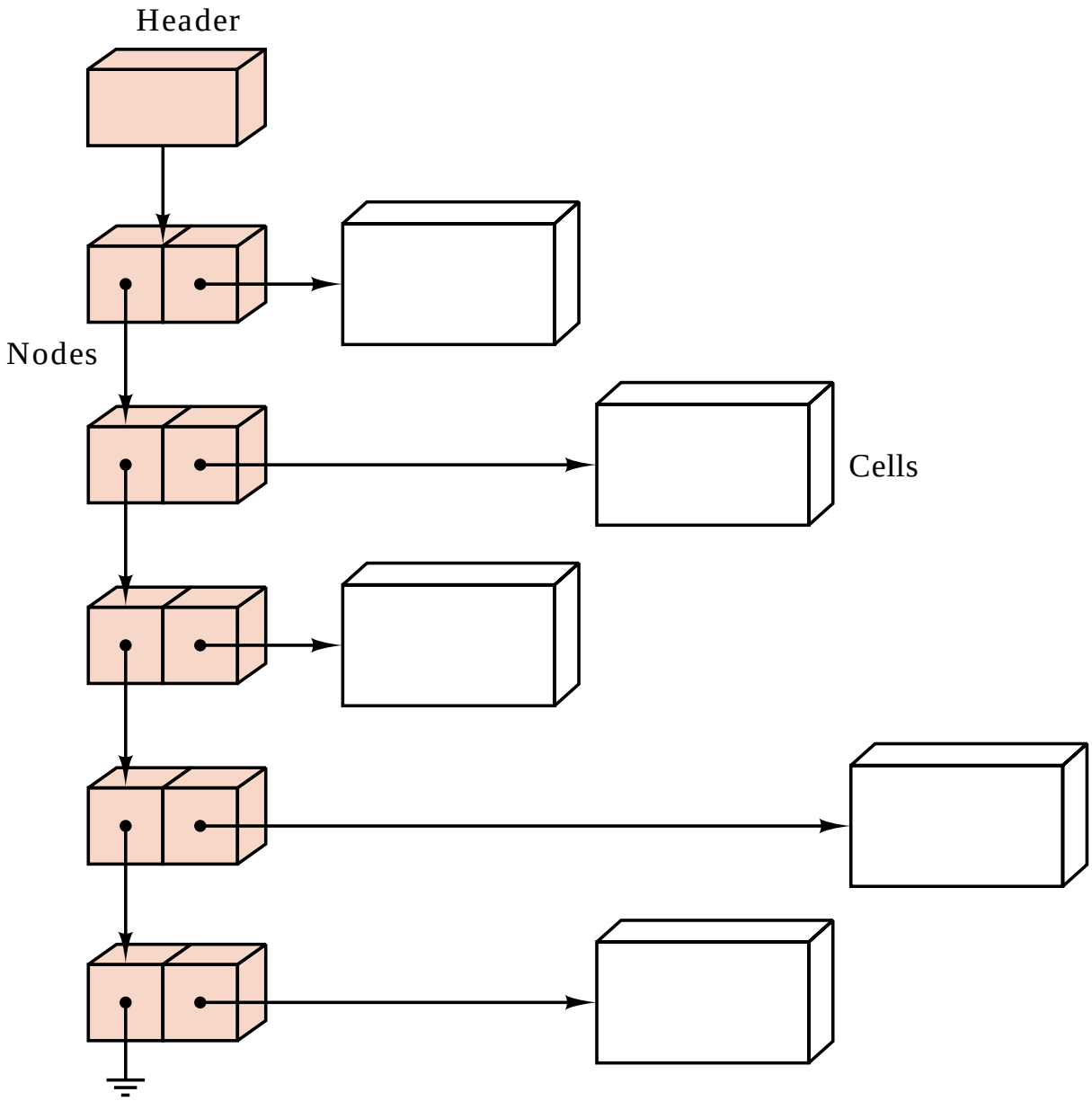
Data Structures for Life

- A Life configuration includes a hash table to look up the status of cells.
- We need to traverse the living cells in a configuration. To do this, we keep a List of living cells as a second data member of a Life configuration.
- An object stored in the list and table is a structure called Cell, containing a pair of grid coordinates:

```
struct Cell {
   Cell( ) { row = col = 0; }   //    constructors
   Cell(int x, int y) { row = x;  col = y;  }
   int row, col;                //    grid coordinates
};
```

- As a Life configuration expands, cells on its fringes will be encountered for the first time. Whenever a new Cell is needed, it must be created dynamically, so Cell objects will only be accessed through pointers.
- To dispose of a Cell object, at the end of its lifetime, we must remember the corresponding pointer. Therefore, the List member will store pointers to cells rather than the cells themselves. This is called an ***indirect list***.

Indirect Linked List of Cells



- Each node of the List contains two pointers: one to a Cell and one to the next Node of the List.

The Hash Table

- Entries of the hash table will be pointers to cells, as in the List.
- The coordinates of the cells, which are determined by the pointers, are the corresponding keys.
- We must decide between open addressing and chaining.
- The entries to be stored in the table need little space: Each entry need only store a pointer to a Cell. Since the table entries are small, there are few space considerations to advise our decision.
- With chaining, the size of each record will increase 100 percent to accommodate the necessary pointer, but the hash table itself will be smaller and can take a higher load factor than with open addressing.
- With open addressing, the records are smaller, but more room must be left vacant in the hash table to avoid long searches and possible overflow.
- For flexibility, let us decide to use a chained hash table.

```
class Hash_table {
public:
   Error_code insert(Cell *new_entry);
   bool retrieve(int row, int col) const;
private:
   List<Cell *> table[hash_size];
};
```

- A Hash_table comes with a default constructor and destructor, which we shall rely on.

The Life Class

- In the Life class, to facilitate the replacement of a configuration by an updated version, we shall store the data members indirectly, as pointers.

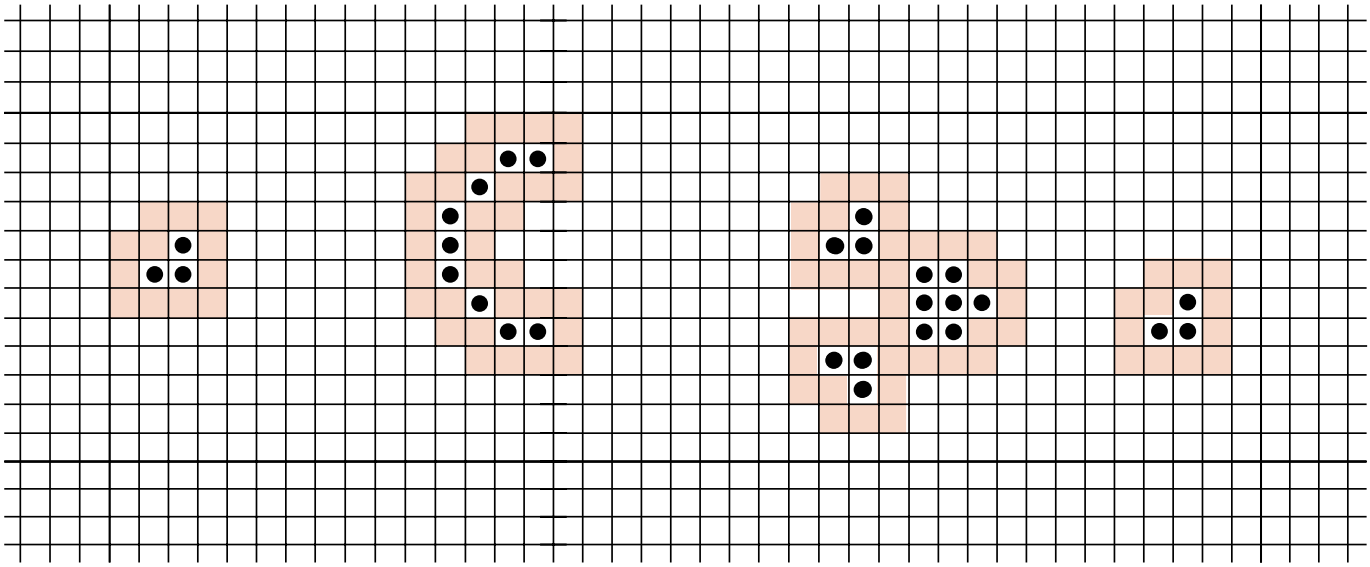
```
class Life {
public:
   Life( );
   void initialize( );
   void print( );
   void update( );
    ~Life( );

private:
   List<Cell *> *living;
   Hash_table *is_living;
   bool retrieve(int row, int col) const;
   Error_code insert(int row, int col);
   int neighbor_count(int row, int col) const;
};
```

- The auxiliary member functions retrieve and neighbor_count determine the status of a cell by applying hash-table retrieval.

- Auxiliary function insert creates a dynamic Cell object and inserts it into both the hash table and the list of cells of a Life object.

Updating the Configuration

- update starts with one Life configuration and determines the configuration at the next generation.
- With an unbounded grid, we must limit our attention to the cells that may be alive in the coming generation. These include the living cells and the dead cells with living neighbors.



- In the method update, a local variable Life new_configuration is thereby gradually built up to represent the upcoming configuration.
- We loop over all the (living) cells from the current configuration, and we also loop over all the (dead) cells that are neighbors of these (living) cells.
- At the end of update, we swap the List and Hash_table members between the current configuration and new_configuration. This exchange also ensures that the destructor that will automatically be applied to the local variable Life new_configuration.

```
void Life :: update( )
/* Post:  The Life object contains the next generation of configuration.
   Uses:  The class Hash_table and the class Life and its auxiliary functions.  */
{
   Life new_configuration;
   Cell *old_cell;
   for (int i = 0;  i < living->size( );  i++) {
      living->retrieve(i, old_cell);   //     Obtain a living cell.
      for (int row_add = -1;  row_add < 2;  row_add ++)
         for (int col_add = -1;  col_add < 2;  col_add++) {
            int new_row = old_cell->row + row_add,
                new_col = old_cell->col + col_add;
//    new_row, new_col is now a living cell or a neighbor of a living cell,
            if (!new_configuration.retrieve(new_row, new_col))
               switch (neighbor_count(new_row, new_col)) {
               case 3:                  //     With neighbor count 3, the cell becomes alive.
                  new_configuration.insert(new_row, new_col);  break;
               case 2:                  //     With count 2, cell keeps the same status.
                  if (retrieve(new_row, new_col))
                     new_configuration.insert(new_row, new_col);  break;
               default:  break;         //     Otherwise, the cell is dead.
               }
         }
   }
//    Exchange data of current configuration with data of new_configuration.
   List<Cell *> *temp_list = living;
   living = new_configuration.living;
   new_configuration.living = temp_list;
   Hash_table *temp_hash = is_living;
   is_living = new_configuration.is_living;
   new_configuration.is_living = temp_hash;
}
```

Printing:

```
void Life :: print( )
/* Post:  A central window onto the Life object is displayed.
   Uses:  The auxiliary function Life :: retrieve.  */
{
   int row, col;
   cout << endl << "The current Life configuration is:" << endl;
   for (row = 0;  row < 20;  row++) {
      for (col = 0;  col < 80;  col++)
         if (retrieve(row, col)) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Creation and insertion of new cells:

```
Error_code Life :: insert(int row, int col)
/* Pre:   The cell with coordinates row and col is not in the Life configuration.
   Post:  The cell has been added to the configuration.  If insertion into either the
          List or the Hash_table fails, an error code is returned.
   Uses:  The class List, the class Hash_table, and the struct Cell  */
{
   Error_code outcome;
   Cell *new_cell = new Cell(row, col);
   int index = living->size( );
   outcome = living->insert(index, new_cell);
   if (outcome ==  success)
      outcome = is_living->insert(new_cell);
   if (outcome != success)
      cout << " Warning: new Cell insertion failed" << endl;
   return outcome;
}
```

Construction and destruction of Life objects:

```
Life :: Life( )
/* Post:  The members of a Life object are dynamically allocated and initialized.
   Uses:  The class Hash_table and the class List.  */
{ living = new List<Cell *>;  is_living = new Hash_table;  }

Life :: ~Life( )
/* Post:  The dynamically allocated members of a Life object and all Cell objects
          that they reference are deleted.
   Uses:  The class Hash_table and the class List.  */
{
   Cell *old_cell;
   for (int i = 0;  i < living->size( );  i++) {
      living->retrieve(i, old_cell);
      delete old_cell;
   }
   delete is_living;          //   Calls the Hash_table destructor
   delete living;             //   Calls the List destructor
}
```

The hash function:

```
const int factor = 101;

int hash(int row, int col)
/* Post:  The function returns the hashed valued between 0 and hash_size - 1
          that corresponds to the given Cell parameter.  */
{
   int value;
   value = row + factor * col;
   value %= hash_size;
   if (value < 0) return value + hash_size;
   else return value;
}
```

Pointers and Pitfalls

1. Use top-down design for your data structures, just as you do for your algorithms.

2. Before considering detailed structures, decide what operations on the data will be required.

3. For the design and programming of lists, see Chapter 6.

4. Use the logical structure of the data to decide what kind of table to use.

5. Let the structure of the data help you decide whether an index function or an access table is better for accessing a table of data.

6. In using a hash table, let the nature of the data and the required operations help you decide between chaining and open addressing.

7. Hash functions must usually be custom-designed for the kind of keys used for accessing the hash table.

8. Recall from the analysis of hashing that some collisions will almost inevitably occur.

9. For open addressing, clustering is unlikely to be a problem until the hash table is more than half full.