Data Structures and Algorithms (12)

Instructor: Ming Zhang
Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/
Chapter 12 Advanced Data Structure

• 12.1 Multidimensional array
  • 12.1.1 Basic Concepts
  • 12.1.2 Structure of Array
  • 12.1.3 Storage of Array
  • 12.1.4 Declaration of Array
  • 12.1.5 Special Matrices Implemented by Arrays
  • 12.1.6 Sparse Matrix
• 12.2 Generalized List
• 12.3 Storage management
• 12.4 Trie
• 12.5 Improved BST
Basic Concepts

• Array is an ordered sequence with fixed number of elements and type.
• The size and type of static array must be specified at compile time.
• Dynamic array is allocated memory at runtime.
Basic Concepts

• Multidimensional array is an extension of one-dimensional array (vector).
• Vector of vectors make up an multidimensional array.
• Represented as
  \[ \text{ELEM A}[c_1..d_1][c_2..d_2]...[c_n..d_n] \]
• \( c_i \) and \( d_i \) are upper and lower bounds of the indices in the i-th dimension. Thus, the total number of elements is:
  \[ \prod_{i=1}^{n} (d_i - c_i + 1) \]
12.1 Multidimensional Array

Structure of Array

2-dimensional array

d1[0..2], d2[0..3], d3[0..1] are the three dimensions respectively

3-dimensional array

d1=3, d2=4, d3=2

d1=3, d2=5

Storage of Array

- Memory is one-dimensional, so arrays are stored linearly
  - Stored row by row (row-major)
  - Stored column by column (column-major)

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
\]
Chapter 12

Advanced Data Structure

12.1 Multidimensional Array

Row-Major in Pascal

\[
\begin{array}{cccc}
  a_{111} & a_{112} & a_{113} & \cdots & a_{11n} \\
  a_{121} & a_{122} & a_{123} & \cdots & a_{12n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{1m1} & a_{1m2} & a_{1m3} & \cdots & a_{1mn} \\
  a_{211} & a_{212} & a_{213} & \cdots & a_{21n} \\
  a_{221} & a_{222} & a_{223} & \cdots & a_{22n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{2m1} & a_{2m2} & a_{2m3} & \cdots & a_{2mn} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{k11} & a_{k12} & a_{k13} & \cdots & a_{k1n} \\
  a_{k21} & a_{k22} & a_{k23} & \cdots & a_{k2n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{km1} & a_{km2} & a_{km3} & \cdots & a_{kmn} \\
\end{array}
\]

\[a[1..k,1..m,1..n]\]
12.1 Multidimensional Array

**Column-Major in FORTRAN**  
\[ a[1..k, 1..m, 1..n] \]

\[
\begin{array}{cccc}
  a_{11} & a_{21} & a_{31} & \ldots \ a_{k1} \\
  a_{12} & a_{22} & a_{32} & \ldots \ a_{k2} \\
  \vdots \\
  a_{1m} & a_{2m} & a_{3m} & \ldots \ a_{km} \\
\end{array}
\]

\[
\begin{array}{cccc}
  a_{11} & a_{12} & a_{13} & \ldots \ a_{1n} \\
  a_{21} & a_{22} & a_{23} & \ldots \ a_{2n} \\
  \vdots \\
  a_{m1} & a_{m2} & a_{m3} & \ldots \ a_{mn} \\
\end{array}
\]
• C++ multidimensional array
  
  ELEM A[d_{1}][ d_{2}][...][d_{n}];

  \[ loc(A[j_1, j_2, \ldots, j_n]) = loc(A[0, 0, \ldots, 0]) + d \cdot [j_1 \cdot d_2 \cdot \ldots \cdot d_n + j_2 \cdot d_3 \cdot \ldots \cdot d_n + \ldots + j_{n-1} \cdot d_n + j_n] \]

  \[ = loc(A[0, 0, \ldots, 0]) + d \cdot \left[ \sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} d_k + j_n \right] \]
Special Matrices Implemented by Arrays

- Triangular matrix (upper/lower)
- Symmetric matrix
- Diagonal matrix
- Sparse matrix
Lower Triangular Matrix

• One-dimensional array: list[0.. (n²+n)/2-1]
  • The matrix element $a_{i,j}$ is stored in list[ (i²+i) /2 + j] (i≥j)

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
7 & 5 & 0 & 0 \\
0 & 0 & 1 & 0 \\
9 & 0 & 0 & 1 & 8 \\
0 & 6 & 2 & 2 & 0 & 7
\end{bmatrix}
\]
Symmetric Matrix

- Satisfies that $a_{i,j} = a_{j,i}$, $0 \leq i, j < n$
  The matrix on the right is a (symmetric) adjacent matrix for a undirected graph

- Store the lower triangle in a 1-dimensional array
  
  $sa[0..n(n+1)/2-1]$
  
  - There is a one-to-one mapping between $sa[k]$ and $a_{i,j}$:
    
    $$k = \begin{cases} 
    j(j + 1)/2 + i, & i < j \\
    i(i + 1)/2 + j, & i \geq j 
    \end{cases}$$
Diagonal Matrix

- Diagonal matrix: all non-zero elements are located at diagonal lines.

- Band matrix: $a[i][j] = 0$ when $|i-j| > 1$
  
  - A band matrix with bandwidth 1 is shown as below

\[
\begin{pmatrix}
 a_{0,0} & a_{0,1} & & & & 0 \\
 a_{1,0} & a_{1,1} & a_{1,2} & & & \\
 & a_{1,0} & a_{1,1} & a_{1,2} & & \\
 & & & \ddots & & \\
 0 & & & & a_{n-1,n-2} & a_{n-1,n-1} \\
 a_{0,1} & & & & a_{n-2,n-1} & a_{n-1,n-1}
\end{pmatrix}
\]
Sparse Matrix

- Few non-zero elements, and these elements distribute unevenly

\[
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]
• Sparse Factor
  • In a $m \times n$ matrix, there are $t$ non-zero elements, and the sparse factor is:
    \[ \delta = \frac{t}{m \times n} \]
  • When this value is lower than 0.05, the matrix could be considered a sparse matrix.

• 3-tuple $(i, j, a_{ij})$: commonly used for input/output
  • $i$ is the row number
  • $j$ is the column number
  • $a_{ij}$ is the element value
Orthogonal Lists of a Sparse Matrix

- An orthogonal list consists of two sets of lists
  - pointer sequence for rows and columns
  - Each node has two pointers: one points to the successor on the same row; the other points to the successor on the same column

\[
\begin{bmatrix}
0 & 3 & 0 \\
0 & 5 & 6 \\
2 & 0 & 0
\end{bmatrix}
\]
Classic Matrix Multiplication

- $A[c_1..d_1][c_3..d_3]$, $B[c_3..d_3][c_2..d_2]$, $C[c_1..d_1][c_2..d_2]$.

$$C = A \times B \quad (C_{ij} = \sum_{k=c_3}^{d_3} A_{ik} \cdot B_{kj})$$
Time Cost of Classic Matrix Multiplication

• p = d1 - c1 + 1, m = d3 - c3 + 1, n = d2 - c2 + 1;
• A is a p×m matrix, B is a m×n matrix, resulting in C, a p×n matrix
• So the time cost of the classic matrix multiplication is O(p×m×n)

```c
for (i = c1; i <= d1; i++)
    for (j = c2; j <= d2; j++){
        sum = 0;
        for (k = c3; k <= d3; k++)
            sum = sum + A[i,k]*B[k,j];
        C[i,j] = sum;
    }
```
12.1  Multidimensional Array

Sparse Matrix Multiplication

\[
\begin{bmatrix}
3 & 0 & 0 & 5 \\
0 & -1 & 0 & 0 \\
2 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
0 & 2 \\
1 & 0 \\
-2 & 4
\end{bmatrix}
= 
\begin{bmatrix}
0 & 6 \\
6 & 0 \\
-1 & 4
\end{bmatrix}
\]
Time Cost of Sparse Matrix Multiplication

• A is a $p \times m$ matrix, $B$ is a $m \times n$ matrix, resulting in $C$, a $p \times n$ matrix.
  • If the number of non-zero elements in a row of $A$ is at most $t_a$
  • and the number of non-zero elements in a column of $B$ is at most $t_b$
• Overall running time is reduced to $O((t_a + t_b) \times p \times n)$
• Time cost of classic matrix multiplication is $O(p \times m \times n)$
Applications of Sparse Matrix

polynomial of one indeterminate

\[ P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \]

\[ = \sum_{i=0}^{n} a_i x^i \]
Chapter 12 Advanced Data Structure

• 12.1 Multi-array
• 12.2 Generalized List
  • Basic Concepts
  • Different Types of Generalized List
  • Storage of Generalized List
  • Traversal algorithm for Generalized List
• 12.3 Storage management
• 12.4 Trie
• 12.5 Improved BST
Basic Concepts

• Review of linear list
  - Finite ordered sequence consisting of \( n(\geq 0) \) elements.
  - All elements of a linear list have the same type.

• If a linear list contains one or more sub-lists, then it is called a generalized list, usually represented as:
  - \( L = (x_0, x_1, ..., x_i, ..., x_{n-1}) \)
\[ L = (x_0, x_1, \ldots, x_i, \ldots, x_{n-1}) \]

- \( L \) is the **name** of this generalized list.
- \( n \) is the **length**.
- Each \( x_i(0 \leq i \leq n-1) \) is an **element**.
  - either a single element, i.e. atom,
  - or another generalized list, i.e. sublist.
- **Depth**: the number of brackets when all the elements are converted to atoms.
L = (x_0, x_1, ..., x_i, ..., x_{n-1})

• head = x_0
• tail = (x_1, ..., x_{n-1})
  • smaller lists
• Easier to store and to implement.
Different Types of Generalized Lits

• pure list
  • There is only one path existing from root to each leaf.
  • i.e. each element (atom, sublist) only appears once. 

\((x_1, (y_1, (a_1, a_2), y_3), x_3, (z_1, z_2))\)
Different Types of Generalized Lits

- **Reentrant lists**
  - Its elements (atoms or sublists) might appear more than once.
  - Corresponds to a DAG if no circles exist.

- **Sublists and atoms are labeled.**

**Example cycle lists**

\[
\left( (a, b) \right), \left( (a, b) , c, d \right), \left( d, e, f, g \right), \left( f, g \right)
\]

- **Diagram:**

```
          L3
          /\  \
        /   \  
L1   L2    L3
  |   |     |   |
 a   b   c   d   e   f   g
```

\[
(L1: (a,b), \ (L1, c ,L2: (d) ), \ (L2, e,L3: (f,g) ), \ L3)
\]
Different Types of Generalized Lits

• Circle lists
  • contains circles.
  • with infinite depth.

(L1: (L2: (L1, a) ) , (L2, L3: (b) ) , (L3, c) , L4: (d,L4) )
Chapter 12
Advanced Data Structure

12.2 Generalized list and Storage management

A                B                 C                                   D

E                B       C          A

Linear Lists

6      2

Pure Lists

5      3    'x'

Reentrant Lists

6      2

Circle Lists

4

Ming Zhang  “Data Structures and Algorithms”
• Graph ⊇ Reentrant List ⊇ Pure List(Tree) ⊇ Linear List
  • Generalized lists are extensions of linear and tree structures.
• Circle lists are reentrant lists that have circles.
• Applications of generalized lists
  • Relations between the invocation of the function
  • Reference relations in memory space
  • LISP
Storage of Generalized Lists

- Generalized link lists without head node
  - Problems might occur when deleting nodes.
  - The list must be adjusted when deleting node 'data'.

![Diagram of generalized list storage](image)
Storage of Generalized Lists

- Add the head node, and the deleting/inserting operation would be simplified.
- Reentrant lists, especially circle lists
  - mark each node (because it is a graph)
Circle Generalized Lists with Head Nodes

Chapter 12
Advanced Data Structure

12.2 Generalized list and Storage management
Chapter 12
Advanced Data Structure

12.2 Generalized list and Storage management

(L1: (L2 : (a,L1 ))

Diagram showing a generalized list structure with nodes labeled with variables and arrows indicating connections.
12.2 Generalized list and Storage management

\[(L_1: (L_2: (a, L_1))) \quad \text{, } L_x : (L_2, L_3 : (b)) \quad \text{, } L_y : (L_3, c) \quad \text{, } L_4 : (d, L_4))\]
Chapter 12 Advanced Data Structure

• 12.1 Multidimensional array
• 12.2 Generalized Lists
• 12.3 Storage management
  • Allocation and Reclamation
  • Freelist
  • Dynamic Memory Allocation and Reclamation
  • Failure Policy and Collection of Useless Units
• 12.4 Trie
• 12.5 Improved BST
Allocation and Reclamation

- Basic problems in storage management
  - Allocate memory
  - Reclaim "freed" memory
- Fragmentation problem
  - The compression of storage
- Collection of useless units
  - Useless units: memory that can be collected but has not been collected yet
  - Memory leak
    - Programmers forget to delete pointers which will not be used
Freelist

- Consider the memory as an array of changeable number of blocks
  - Some blocks has been allocated
  - Link free blocks together, and form a freelist.
- Memory allocation and reclamation
  - new p: allocate from available space
  - delete p: return the block that p points to to the freelist.
- If there is not enough space, resort to failure policy.
Chapter 12
Advanced Data Structure

12.3 Storage Management

(1) initial state of the freelist

freelist with nodes of equal length

(2) freelist after the system has run for a while
Function overloading of freelist

```cpp
template <class Elem> class LinkNode{
private:
    static LinkNode avail; // head pointer
public:
    Elem value; // value of each node
    LinkNode   next; // pointer pointing to next node
    LinkNode (const Elem & val, LinkNode p); // construction function
    LinkNode (LinkNode p = NULL); // construction function
    void operator new (size_t); // redefine new
    void operator delete (void p); // redefine delete
};
```
// implementation of new

template <class Elem>
void LinkNode<Elem>::operator new (size_t) {
    if (avail == NULL) // if the list is empty
        return ::new LinkNode; // allocate memory using new
    LinkNode<Elem> temp = avail;

    // allocate from available space
    list
    avail = avail->next;
    return temp;
}
//implementation of delete

template <class Elem>
void LinkNode<Elem>::operator delete (void * p) {
    ((LinkNode<Elem> *) p)->next = avail;
    avail = (LinkNode<Elem> *) p;
}
Free List: Stack in a Singly-Linked List

- new: deletion in the stack
- delete: insertion in the stack
- If the default new and delete operations are needed, use “::new p” and “::delete p”.
  - For example, when a program is finished, return the memory occupied by avail back to the system (free the memory completely)
• When $p_{\text{max}}$ is equal to or larger than $S$, no more memory can be allocated.
Dynamic Memory Allocation and Reclamation

Available blocks with variable lengths

• Allocation
  • Find a block whose length is larger than the requested length.
  • Truncate suitable length from it.

• Reclamation
  • Consider whether the space deleted can be merged with adjacent nodes,
  • So as to satisfy later request of large node.
Data Structure of Free Blocks

(a) structure of free block

(b) structure of allocated block
Fragmentation Problem

- Internal fragment: space larger than the requested bytes
- External fragment: small free blocks
Sequential Fit

Allocation of free blocks

• Common sequential fit algorithms
  • first fit
  • best fit
  • worst fit
## Sequential Fit

- **3 Blocks 1200, 1000, 3000**
  - request sequence: 600, 500, 900, 2200

- **first fit:**

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>1200</th>
<th>1000</th>
<th>3000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>600</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>100</td>
<td>2200</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Sequential Fit

• best fit

request sequence: 600, 500, 900, 2200
Sequential Fit

• worst fit

1200  1000  3000

2200  600  500  900  1000

Why always me? ......

request sequence: 600, 500, 900, 2200
Reclamation: merge adjacent blocks

allocate block M back to the freelist
Fitting Strategy Selection

• Need to take the following user request into account
  • Importance of allocation and reclamation efficiency.
  • Variation range of the length of allocated memory
  • Frequency of allocation and reclamation
• In practice, fist fit is the most commonly used.
  • Quicker allocation and reclamation.
  • Support random memory requests.
Hard to decide which one is the best in general.
Failure Policy and Collection of Useless Units

- If a memory request cannot be satisfied because of insufficient memory, the memory manager has two options:
  - do nothing, and return failure info;
  - follow failure policy to satisfy requests.
Compaction

- Collect all the fragments together
  - Generate a larger free block.
  - Used when there are a lot of fragments.
- Handler makes the address relative
  - Secondary indirect reference to the storage location.
- Only have to change handlers to move blocks.
  - No need to change applications.
Two Types of Compaction

• Perform a compact once a block is freed.
• Perform a compact when there is not enough memory or when collecting useless units.

eg:

Before

After
Collecting Useless Units

• Collecting useless units: the most complete failure policy.
  • Search the whole memory, and label those nodes not belonging to any link.
  • Collect them to the freelist.
  • The collection and compaction processes usually can perform at the same time.
Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkjpk/course/sjjg/
Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)