Measuring Input size

Last lecture recap.

Linear data structures

- Compiled from
  
  http://www.eagle.tamut.edu/faculty/igor/CIS-305.htm

Abstract Data Type

Data Structures.
Main Notions and Definitions.

Data Structure

- A **Data Structure** is an aggregation of atomic and composite data into a set with defined relationships.
- **Structure** means a set of rules that holds the data together.
- Taking a combination of data and fit them into such a structure that we can define its relating rules, we create a **data structure**.

Composite Data Structures

<table>
<thead>
<tr>
<th>Array</th>
<th>Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous sequence of data or data types known as elements</td>
<td>Heterogeneous combination of data into a single structure with an identified key</td>
</tr>
<tr>
<td>Position association among the elements</td>
<td>No association</td>
</tr>
</tbody>
</table>

**TABLE 1-2** Data Structure Examples

Data Structure is:

- A combination of elements in which each is either a data type or another data structure
- A set of associations or relationships involving the combined elements
The Abstract Data Type (ADT)

The concept of abstraction means:

- We know what a data type can do
- How it is done is hidden for the user

With an ADT users are not concerned with how the task is done but rather with what it can do.

The Abstract Data Type (ADT)

• A data declaration packaged together with the operations that are meaningful for the data type.
• In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

- Declaration of data
- Declaration of operations
- Encapsulation of data and operations

The Abstract Data Type (ADT)

• All references to and manipulation of the data in a structure must be handled through defined interfaces to the structure.
• Allowing the application program to directly reference the data structure is a common fault in many implementations.
• It is necessary for multiple versions of the structure to be able to coexist.
• We must hide the implementation from the user while being able to store different data.

ADT: Example

• The program code to read/write some data is ADT. It has a data structure (character, array of characters, array of integers, array of floating-point numbers, etc.) and a set of operations that can be used to read/write that data structure.
ADT Operations

• Data are entered, accessed, modified and deleted through the external interface, which is a “passageway” located partially “in” and partially out of the ADT.
• Only the public functions are accessible through this interface.
• For each ADT operation there is an algorithm that performs its specific task.

Typical ADTs:

• Lists
• Stacks
• Queues
• Trees
• Heaps
• Graphs

Linked Lists

• A Linked List is an ordered collection of data in which each element contains the location of the next element or elements.
• In a linked list, each element contains two parts: data and one or more links.
• The data part holds the application data – the data to be processed.
• Links are used to chain the data together. They contain pointers that identify the next element or elements in the list.

Array Implementations

• In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes).
• Although searching an array for an individual element can be very efficient, insertion and deletion of elements are complex and inefficient processes.

Nodes

• A node is a structure that has two parts: the data and one or more links.
• The nodes in a linked list are called self-referential structures. In such a structure, each instance of the structure contains one or more pointers to other instances of the same structural type.

Arrays

• The data part in a node can be a single field, multiple fields, or a structure that contains several fields, but it always acts as a single field.
Linked Lists vs. Arrays

- The major advantage of the linked list over the array is that data are easily inserted and deleted.
- It is not necessary to shift elements of a linked list to make room for a new element.
- However, because the elements are no longer physically sequenced in a linked list, we are limited to sequential searches.

Factorial – a case study

- The factorial of a positive number is the product of the integral values from 1 to the number:

\[ n! = 1 \cdot 2 \cdot 3 \cdots n = \prod_{i=1}^{n} i \]

Factorial: Iterative Algorithm

\[
\text{Factorial}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
rx (n-1) & \text{if } n > 0 
\end{cases} 
\]

Recursion is a repetitive process in which an algorithm calls itself.

Usually recursion is organized in such a way that a subroutine calls itself or a function calls itself.

Two approaches to writing repetitive algorithms:

- Iteration
- Recursion

RECURSION
Factorial: Recursive Algorithm

\[
\text{Factorial}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{Factorial}(n-1) & \text{if } n > 0 
\end{cases}
\]

A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.

Recursion: basic point

- The recursive solution for a problem involves a two-way journey:
- First we decompose the problem from the top to the bottom
- Then we solve the problem from the bottom to the top.

Designing recursive algorithms

- The rules for designing a recursive algorithm:
  1. First, determine the base case.
  2. Then determine the general case.
  3. Combine the base case and the general cases into an algorithm.
Designing recursive algorithms

• Each recursive call must reduce the size of the problem and move it toward the base case.
• The base case, when reached, must terminate without a call to the recursive algorithm; that is, it must execute a return.

Limitations of Recursion

• Recursion should not be used if the answer to any of the following questions is no:
  • Is the algorithm or data structure naturally suited to recursion (tree is the first choice)?
  • Is the recursive solution shorter and more understandable?
  • Does the recursive solution run within acceptable time and space limits?
  • As a general rule, recursive algorithms should be effectively used only when their efficiency is logarithmic.

5-1 Basic Operations

We begin with a discussion of the basic list operations. Each operation is developed using before and after figures to show the changes.

• Insertion
• Deletion
• Retrieval
• Traversal

List

Objectives

Upon completion you will be able to:

• Explain the design, use, and operation of a linear list
• Implement a linear list using a linked list structure
• Understand the operation of the linear list ADT
• Write application programs using the linear list ADT
• Design and implement different link-list structures
Figure 5.3: Retrieval

Figure 5.4: Linked List Implementation of a List

Algorithm 5.1: Create List

Algorithm createList (list)
Pre: list is a linked list structure passed by reference
Post: metadata initialized
1. allocate (list)
2. set list head to nil
3. set list count to 0
end createList

Figure 5.5: Head Node and Data Node

Figure 5.7: Add Node to Empty List

(a) Before add

(b) After add
Algorithm 5.2: Insert Node

Algorithm insertNode(list, pPre, dataIn)

1. Insert data into a new node in the list.
2. pPre = list is pointer to data's logical predecessor
3. pNew = list contains data to be inserted
4. Return true if successful, false if memory overflow
5. Allocate (pNew)
6. Set pNew to dataIn
7. Set pNew to list
8. Set list head to pNew
9. Adding in middle or at end.
10. Set pNew to pPre link
11. Set pPre link to pNew
12. Return true
13. end insertNode

Figure 5.8: Add Node at Beginning

(a) Before add

1
count
head
pNew

(b) After add

Figure 5.9: Add Node in Middle

(a) Before add

2
count
head
pPre
pNew

(b) After add

Figure 5.10: Add Node at End

(a) Before add

3
count
head
pPre
pNew

(b) After add

Figure 5.11: Delete First Node

(a) Before delete

3
count
head

(b) After delete

Figure 5.12: Delete General Case

(a) Before delete

3
count
head

(b) After delete

Figure 5.13: Delete General Case
List Search

A list search is used to locate data in a list.
To insert data, we need to know the logical predecessor to the new data.
To delete data, we need to find the node to be deleted and identify its logical predecessor.
To retrieve data from a list, we need to search the list and find the data.

We must use a sequential search because there is no physical relationship among the nodes.
The classic sequential search returns the location of an element when it is found and the address of the last element when it is not found.
Because the list is ordered, we need to return the location of the element when it is found and the location where it should be placed when it is not found.

<table>
<thead>
<tr>
<th>Condition</th>
<th>pPre</th>
<th>pLoc</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target &lt; first node</td>
<td>Null</td>
<td>first node</td>
<td>False</td>
</tr>
<tr>
<td>Target = first node</td>
<td>Null</td>
<td>first node</td>
<td>True</td>
</tr>
<tr>
<td>Target &lt; target last</td>
<td>Largest node &lt; target</td>
<td>First node &gt; target</td>
<td>False</td>
</tr>
<tr>
<td>Target = target last node</td>
<td>Node’s predecessor</td>
<td>Equal node</td>
<td>True</td>
</tr>
<tr>
<td>Target &gt; target last node</td>
<td>Last node</td>
<td>Last node</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 5-1 List Search Results
Traversals

- Algorithms that **traverse** a list start at the first node and examine each node in succession until the last node has been processed.
- **Traversal logic** is used by such types of algorithms as changing a value in each node, printing a list, summing a list, calculating the average, etc.
- To traverse a list, we need a **walking pointer** that moves from node to node.

Doubly Linked List

- A **doubly linked list** is a linked list structure in which each node has a pointer to both its successor and its predecessor.

Definition

- A **queue** is a linear list in which data can only be inserted at one end, called the **rear**, and deleted from the other end, called the **front**.
- Hence, the data are processed through the queue in the order in which they are received (first in → first out – **FIFO**)

Queues

- A **queue** is a linear list in which data can only be inserted at one end, called the **rear**, and deleted from the other end, called the **front**.
- Hence, the data are processed through the queue in the order in which they are received (first in → first out – **FIFO**).
ALGORITHM 4.1

Create Queue

Algorithm createQueue
Pre: queue is a metadata structure
Post: metadata elements have been initialized
1. Return queue head
2. Allocate queue head
3. Set queue front to null
4. Set queue rear to null
5. Set queue count to 0
6. Return queue head
End createQueue

ALGORITHM 4.2

Insert Data into Queue

Algorithm enqueue(queue, dataIn)
This algorithm inserts data into a queue.
Pre: queue is a metadata structure
Post: dataIn has been inserted
1. If queue full
   1. Return false
   2. End if
2. Allocate new node
3. Move dataIn to new node data
4. Set new node next to null pointer
5. If (empty queue)
   1. Insert into null queue
   2. Set queue front to address of new data
   3. Else
      1. Point old rear to new node
      2. Set next pointer of rear node to address of new node
      3. End if
6. Set queue rear to address of new node
7. Increment queue count
8. Return true
End enqueue

ALGORITHM 4.3

Delete Data from Queue

Algorithm dequeue(queue, item)
This algorithm deletes a node from a queue.
Pre: queue is a metadata structure
Post: data at queue front returned to user through item
and front element deleted
1. Return true if successful, false if underflow
2. If (queue empty)
   1. Return false
   2. End if
3. Move front data to item
STACKS

Stack

- A stack is a linear list in which additions and deletions of data are restricted to one end, called the top.
- If we insert a data series into a stack and then remove it, the order of data is reversed.
- This property is known as the last in – first out.
FIGURE 3-6  Conceptual and Physical Stack Implementations

(a) Conceptual  (b) Physical

FIGURE 3-7  Stack Data Structure

Stack head structure

count  top

Stack node structure

data  link

node  data  link

end node

end stack

FIGURE 3-9  Push Stack Example

(a) Before  (b) After

FIGURE 3-8  Stack Operations

ALGORITHM 3-1  Create Stack

Algorithm createStack

Create and initializes metadata structure.

Pre: Nothing

Post: Structure created and initialized

Return: stack head

1. allocate memory for stack head
2. set count to 0
3. set top to null
4. return stack head

end createStack

FIGURE 3-5  Stack Example
ALGORITHM 3-2 Push Stack Design

Algorithm pushStack (stack, data)

Insert (push) one item into the stack.

Pre stack passed by reference

data contain data to be pushed into stack

Post data have been pushed in stack

1. allocate new node
2. store data in new node
3. make current top node the second node
4. make new node the top
5. increment stack count
end pushStack

ALGORITHM 3-4 Stack Top Pseudocode

Algorithm stackTop (stack, dataOut)

This algorithm retrieves the data from the top of the stack
without changing the stack.

Pre stack is a data structure to a valid stack

dataOut is reference variable to receive data

Post data have been returned to calling algorithm

Return true if data returned, false if underflow

1. if (stack empty)
2. set success to false
3. else
4. set dataOut to data in top node
5. set success to true
6. end if
7. return success
end stackTop

PROGRAM 3-6 Stack ADT Definitions

1 // Stack ADT Type Definitions
2 typedef struct node
3 { 
4 void* dataPtr;
5 struct node* link;
6 } STACK_NODE;
7
8 typedef struct
9 { 
10 int count;
11 STACK_NODE* top;
12 } STACK;

FIGURE 3-10 Pop Stack Example

(a) Before

(b) After

FIGURE 3-12 Stack ADT Structural Concepts
Parse Parentheses

(a) Opening parenthesis not matched

(b) Closing parenthesis not matched

FIGURE 3.14 Unmatched Parentheses Examples

Algorithm 3.9

Algorithm parseparentheses

This algorithm reads a source program and parses it to make sure all opening-closing parentheses are paired.

1. loop (over data)
   1. read (character);
   2. if (opening parenthesis)
      1. pushstack (stack, character);
   3. else
      1. if (closing parenthesis)
         1. if (emptystack (stack))
            1. print (error: closing parentheses not matched)
         2. else
            1. popstack (stack)
      2. end if
   2. end if
3. end loop
4. if (not emptystack (stack))
   1. print (error: opening parenthesis not matched)
5. end parseparentheses

Hash Tables

www.cs.ucf.edu/courses/cot4810/fall04/presentations/Hash_Tables.ppt

COT4810

Ken Pritchard

2 Sep 04

History

• The term hashing was apparently developed through an analogy to compare the way the key would be mangled in a hash function to the standard meaning of hashing being to chop something up.
• 1953 – Hashing with chaining was mentioned in an internal IBM memorandum.
• 1956 – Hashing was mentioned in a publication by Arnold I. Dumey, Computers and Automation
• 1968 – Random probing with secondary clustering was described by Robert Morris in CACM 11
• 1973 – Donald Knuth wrote The Art of Computer Programming in which he describes and analyzes hashing in depth.

Description

• A hash table is a data structure that stores things and allows insertions, lookups, and deletions to be performed in O(1) time.
• An algorithm converts an object, typically a string, to a number. Then the number is compressed according to the size of the table and used as an index.
• There is the possibility of distinct items being mapped to the same key. This is called a collision and must be resolved.
Collision Resolution

- There are two kinds of collision resolution:
  1. Chaining makes each entry a linked list so that when a collision occurs the new entry is added to the end of the list.
  2. Open Addressing uses probing to discover an empty spot.
- With chaining, the table does not have to be resized when the number of elements is larger than the capacity.

Collision Resolution – Chaining

- With chaining, each entry is the head of a (possibly empty) linked list. When a new object hashes to an entry, it is added to the end of the list.
- A particularly bad hash function could create a table with only one non-empty list that contained all the elements. Uniform hashing is very important with chaining.
- The load factor of a chained hash table indicates how many objects should be found at each location. The load factor \( LF = \frac{n}{c} \) where \( n \) is the number of objects stored and \( c \) is the capacity of the table.
- With uniform hashing, a load factor of 2 means we expect to find no more than two objects in one slot. A load factor less than 1 means that we are wasting space.

Collision Resolution – Open Addressing

- With open addressing, the table is probed for an open slot when the first one already has an element.
- There are different types of probing, some more complicated than others. The simplest type is to keep increasing the index by one until an open slot is found.
- The load factor of an open addressed hash table can never be more than 1. A load factor of .5 indicates that the table is half full.
- With uniform hashing, the number of positions that we can expect to probe is \( \frac{1}{1 - LF} \). For a table that is half full, \( LF = .5 \), we can expect to probe \( \frac{1}{1 - .5} = 2 \) positions. Note that for a table that is 95% full, we can expect to probe 20 positions.

Chaining

- Smith → 7
- Bob Smith
  123 Main St.
  Orlando, FL 327816
  407-555-1111
  bob@myisp.com
- Jim Smith
  123 Elm St.
  Orlando, FL 327816
  407-555-2222
  jim@myisp.com

Probing

- Smith → 7
- Bob Smith
  123 Main St.
  Orlando, FL 327816
  407-555-1111
  bob@myisp.com
- Jim Smith
  123 Elm St.
  Orlando, FL 327816
  407-555-2222
  jim@myisp.com
Hash Functions

- Hash Functions perform two separate functions:
  1. Convert the string to a key.
  2. Constrain the key to a positive value less than the size of the table.
- The best strategy is to keep the two functions separate so that there is only one part to change if the size of the table changes.

Hash Functions - Key Generation

- There are different algorithms to convert a string to a key.
- There is usually a trade off between efficiency and completeness.
- Some very efficient algorithms use bit shifting and addition to compute a fairly uniform hash.
- Some less efficient algorithms give a weight to the position of each character using a base related to the ASCII codes. The key is guaranteed unique, but can be very long. Also, some of the precision gained will be lost in compression.

Hash Functions - Compression

- In the multiplication method, the key is multiplied by a constant \( A \) in the range: \( 0 < A < 1 \)
- Extract the fractional part of the result and multiply it by the size of the table to get the index.
- \( A \) is usually chosen to be 0.618, which is approximately: \( (\sqrt{5} - 1)/2 \)
- The entire computation for the index follows:
  \[
  \text{index} = \text{floor} (\text{table size} \times ((\text{key} \times A) \mod 1))
  \]

Hash Functions - Example

```c
int hash(char * key)
{
    int val = 0;
    while(*key != '\0')
    {
        val = (val << 4) + (*key);
        key++;
    }
    return val;
}

int compress(int index, int size)
{
    return abs(index % size);
}
```

Dynamic Modification

- If the total number of items to store in the table are not known in advance, the table may have to be modified. In any event, the insert and delete functions should track the number of items.
- If the table uses chaining, it will not have to be modified unless the user wants to specify a maximum load value so that performance does not deteriorate.
- If the table uses open addressing, the table will have to resized when full before another item can be stored; however, it might be resized before that point based on the load factor so that performance does not deteriorate.
Hash Table Uses

- Compilers use hash tables for symbol storage.
- The Linux Kernel uses hash tables to manage memory pages and buffers.
- High speed routing tables use hash tables.
- Database systems use hash tables.

Example

C Example

C# Example

Java Example

Summary

- A hash table is a convenient data structure for storing items that provides O(1) access time.
- The concepts that drive selection of the key generation and compression functions can be complicated, but there is a lot of research information available.
- There are many areas where hash tables are used.
- Modern programming languages typically provide a hash table implementation ready for use in applications.

References


